

Николай Прохоренок
Владимир Дронов

PRO

**ПРОФЕССИОНАЛЬНОЕ
ПРОГРАММИРОВАНИЕ**

JavaScript и Node.js

**для веб-
разработчиков**

Итераторы и генераторы
Модули
Класс Promise
Работа с файловой системой
Программирование веб-серверов
Объектная модель документа
AJAX и Fetch API
Webpack
Visual Studio Code



Материалы
на www.bhv.ru



Николай Прохоренок
Владимир Дронов

JavaScript и Node.js

для веб-разработчиков

Санкт-Петербург
«БХВ-Петербург»
2022

УДК 004.438JavaScript
ББК 32.973.26-018.1
П84

Прохоренок, Н. А.

П84 JavaScript и Node.js для веб-разработчиков / Н. А. Прохоренок,
В. А. Дронов. — СПб.: БХВ-Петербург, 2022. — 768 с.: ил. —
(Профессиональное программирование)

ISBN 978-5-9775-6847-0

Книга рассказывает о языке программирования JavaScript, разработке на нем как программ общего назначения, выполняющихся в среде Node.js, так и скриптов для веб-страниц. Даны основы JavaScript: типы данных, операторы, работа с числами, строками, датой и временем, массивами, функции, классы (как старого, так и нового синтаксиса), итераторы, генераторы и класс Promise. Объяснена работа с отладчиком, встроенным в редактор Visual Studio Code. Рассказано о модулях, средствах для работы с файловой системой и программирования веб-серверов. Описана объектная модель документа. Рассмотрены средства для работы с элементами веб-страницы, самой страницей и браузером и технология AJAX (в том числе Fetch API), а также готовые программные пакеты для разработки веб-сайтов, в частности Webpack.

Электронный архив на сайте издательства содержит коды всех пронумерованных листингов.

Для веб-разработчиков

УДК 004.438JavaScript
ББК 32.973.26-018.1

Группа подготовки издания:

Руководитель проекта	<i>Евгений Рыбаков</i>
Зав. редакцией	<i>Людмила Гауль</i>
Компьютерная верстка	<i>Ольги Сергиенко</i>
Дизайн серии	<i>Инны Тачиной</i>
Оформление обложки	<i>Карины Соловьевой</i>

"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

ISBN 978-5-9775-6847-0

© ООО "БХВ", 2022
© Оформление. ООО "БХВ-Петербург", 2022

Оглавление

- Введение 15**
 - Структура книги..... 15
 - Типографские соглашения..... 17
- Глава 1. Редактор Visual Studio Code 19**
 - 1.1. Установка VS Code..... 19
 - 1.2. Настройка редактора 23
 - 1.3. Смена цветовой темы, тем значков файлов и продукта 25
 - 1.4. Структура окна редактора..... 27
 - 1.4.1. Главное меню 28
 - 1.4.2. Боковая панель и панель действий 32
 - 1.4.3. Строка состояния 33
 - 1.4.4. Нижняя панель 34
 - 1.4.5. Палитра команд..... 34
 - 1.5. Работа с файлами и каталогами..... 35
 - 1.5.1. Создание и сохранение файла..... 35
 - 1.5.2. Заккрытие вкладки с файлом 36
 - 1.5.3. Открытие файла 37
 - 1.5.4. Открытие каталога..... 37
 - 1.6. Отображение содержимого нескольких файлов одновременно 39
 - 1.7. Live Server: автоматическое обновление веб-страницы 41
 - 1.8. Emmet: ускорение набора HTML-кода 42
 - 1.8.1. Вставка структуры HTML-документа 42
 - 1.8.2. Вставка HTML-комментария 43
 - 1.8.3. Теги из секции заголовка (<head>)..... 43
 - 1.8.4. Основные теги из секции тела (<body>)..... 44
 - 1.8.5. Добавление текста..... 46
 - 1.8.6. Добавление параметров тегов..... 47
 - 1.8.7. Формы и элементы управления 48
 - 1.8.8. Таблицы 49
 - 1.8.9. Списки..... 50
 - 1.8.10. Вставка сразу нескольких тегов..... 51

1.9. Emmet: ускорение набора CSS-кода.....	52
1.9.1. Форматирование шрифта	52
1.9.2. Форматирование текста.....	55
1.9.3. Отступы	57
1.9.4. Рамки.....	58
1.9.5. Фон элемента.....	60
1.9.6. Списки.....	61
1.9.7. Таблицы	62
1.9.8. Вид курсора.....	62
1.9.9. Форматирование блоков.....	62
1.9.10. Flex-контейнеры.....	64
1.9.11. CSS Grid.....	66
1.9.12. Многоколоночный текст	68
1.9.13. Эффекты	68
1.9.14. Анимация с двумя состояниями	69
1.9.15. Анимация с несколькими состояниями.....	69
1.9.16. Двумерные трансформации	70
1.9.17. Трехмерные трансформации.....	71
1.9.18. Прочие атрибуты и правила.....	71
1.9.19. Ввод числового значения атрибута	73
1.9.20. Добавление вендорных префиксов.....	73
1.10. Всплывающие подсказки при вводе.....	74
1.11. Пользовательские фрагменты кода	75
1.11.1. В Emmet.....	75
1.11.2. В редакторе VS Code	77
1.12. Работа с курсорами и выделениями	82
1.13. Дублирование, перемещение и удаление строк	83
1.14. Сворачивание и разворачивание блоков кода	84
1.15. Изменение регистра символов.....	84
1.16. Изменение масштаба	85
1.17. Полноэкранный режим.....	85
1.18. Форматирование кода	85
1.19. Полезные расширения.....	86
Глава 2. Node.js: первые шаги.....	88
2.1. Установка Node.js	88
2.2. Работа с командной строкой.....	92
2.3. Первая программа на JavaScript	94
2.4. Вкладка <i>Терминал</i> в редакторе VS Code.....	95
2.5. NPM (Node Package Manager).....	97
2.5.1. Создание пакета и добавление файла package.json	98
2.5.2. Установка пакетов	101
2.5.3. Обновление и удаление пакетов	103
2.5.4. Добавление и запуск скриптов.....	104
2.6. Структура программы	108
2.7. Комментарии в JavaScript	109
2.8. Интерактивный режим	111
2.9. Вывод результатов работы программы	112

2.10. Получение данных из командной строки	115
2.11. Завершение выполнения программы	117
2.12. Получение версии Node.js.....	117

Глава 3. Переменные и типы данных 119

3.1. Именованние переменных	119
3.2. Объявление переменной	120
3.3. Инициализация переменной и присваивание значения.....	120
3.4. Ключевые слова <i>var</i> и <i>let</i>	121
3.5. Типы данных в языке JavaScript	123
3.6. Проверка существования переменной	124
3.7. Преобразование типов данных	125
3.8. Классы-"обертки" над элементарными типами	127
3.9. Константы	128
3.10. Области видимости переменных	129
3.11. Вывод значений переменных.....	132

Глава 4. Операторы и циклы..... 133

4.1. Математические операторы	133
4.2. Побитовые операторы	135
4.3. Оператор запятая	136
4.4. Операторы сравнения.....	137
4.5. Оператор нулевого слияния	139
4.6. Операторы присваивания.....	139
4.7. Приоритет выполнения операторов	140
4.8. Оператор ветвления <i>if...else</i>	141
4.9. Оператор <i>?:</i>	147
4.10. Оператор выбора <i>switch</i>	148
4.11. Цикл <i>for</i>	151
4.12. Цикл <i>while</i>	153
4.13. Цикл <i>do...while</i>	154
4.14. Цикл <i>for...in</i>	155
4.15. Цикл <i>for...of</i>	157
4.16. Оператор <i>continue</i> : переход на следующую итерацию цикла	157
4.17. Оператор <i>break</i> : прерывание цикла.....	158

Глава 5. Числа..... 159

5.1. Указание значений.....	159
5.2. Класс <i>Number</i>	160
5.3. Математические константы	164
5.4. Основные методы для работы с числами	165
5.5. Округление чисел	166
5.6. Тригонометрические функции	166
5.7. Преобразование строки в число	167
5.8. Преобразование числа в строку.....	168
5.9. Генерация псевдослучайных чисел	171
5.10. Бесконечность и значение <i>NaN</i>	171
5.11. Тип данных <i>bigint</i>	172

Глава 6. Массивы и множества	175
6.1. Инициализация массива	175
6.2. Получение и изменение элемента массива	177
6.3. Определение размера массива	179
6.4. Многомерные массивы	179
6.5. Создание копии массива	181
6.6. Объединение массивов	182
6.7. Перебор элементов массива	182
6.8. Добавление и удаление элементов массива	183
6.9. "Перевооруживание" массива	185
6.10. Сортировка массива	185
6.11. Получение части массива	186
6.12. Преобразование массива	187
6.13. Поиск элемента в массиве	189
6.14. Фильтрация массива	190
6.15. Методы, возвращающие итераторы	191
6.16. Ассоциативные массивы	192
6.17. Класс <i>Map</i> : словарь	193
6.17.1. Создание объекта	193
6.17.2. Добавление элементов и изменение значения	194
6.17.3. Получение значения по ключу	194
6.17.4. Проверка наличия ключа	194
6.17.5. Определение размера массива	194
6.17.6. Удаление элементов	195
6.17.7. Методы класса <i>Map</i> , возвращающие итераторы	195
6.17.8. Перебор элементов словаря	196
6.18. Класс <i>Set</i> : множество	197
6.18.1. Создание объекта	197
6.18.2. Добавление элементов	197
6.18.3. Проверка наличия элемента	198
6.18.4. Определение размера множества	198
6.18.5. Удаление элементов	198
6.18.6. Методы класса <i>Set</i> , возвращающие итераторы	199
6.18.7. Перебор элементов множества	199
6.18.8. Преобразование множества в массив	200
6.18.9. Операции с двумя множествами	200
Глава 7. Строки	202
7.1. Инициализация строк	202
7.2. Специальные символы в строке	204
7.3. Строки в обратных кавычках (шаблоны строк)	205
7.4. Неформатированные строки	206
7.5. Конкатенация строк	206
7.6. Определение длины строки	207
7.7. Обращение к отдельному символу в строке	207
7.8. Изменение регистра символов	208
7.9. Получение фрагмента строки	209

7.10. Сравнение строк	209
7.11. Поиск и замена в строке.....	210
7.12. Преобразование строки в массив	212
7.13. URL-кодирование строк.....	213
7.14. Повтор строки.....	213
7.15. Выполнение команд, содержащихся в строке.....	214

Глава 8. Регулярные выражения 215

8.1. Создание регулярного выражения	215
8.2. Методы класса <i>String</i>	216
8.3. Методы класса <i>RegExp</i>	222
8.4. Свойства класса <i>RegExp</i>	224
8.5. Синтаксис регулярных выражений	226
8.5.1. Экранирование специальных символов	226
8.5.2. Метасимволы.....	228
8.5.3. Стандартные классы	230
8.5.4. Квантификаторы.....	231
8.5.5. "Жадность" квантификаторов	231
8.5.6. Группы	232
8.5.7. Обратные ссылки	235
8.5.8. Просмотр вперед и назад.....	235

Глава 9. Работа с датой и временем.....238

9.1. Получение текущей даты и времени	238
9.2. Указание произвольных значений даты и времени	238
9.3. Разбор строки с датой и временем	239
9.4. Преобразование даты в строку	240
9.5. Получение и изменение значений компонентов даты и времени.....	241
9.6. Форматированный вывод текущей даты и времени	243
9.7. Таймеры.....	244
9.8. Измерение времени выполнения кода	247

Глава 10. Пользовательские функции 249

10.1. Создание функции и ее вызов.....	249
10.2. Расположение определений функций	251
10.3. Вложенные функции	252
10.4. Замыкание	252
10.5. Класс <i>Function</i>	253
10.6. Функции с произвольным числом параметров	255
10.6.1. Объект <i>arguments</i>	255
10.6.2. Оператор <i>REST</i>	256
10.7. Способы передачи параметров в функцию	256
10.8. Необязательные параметры	257
10.9. Анонимные функции	258
10.10. Стрелочные функции (лямбда-выражения).....	260
10.11. Функции-генераторы	261
10.12. Рекурсия	263

Глава 11. Классы и объекты	264
11.1. Объекты	264
11.1.1. Создание объекта с помощью фигурных скобок.....	264
11.1.2. Свойства объекта.....	265
11.1.3. Распаковка объекта	268
11.1.4. Методы объекта.....	270
11.1.5. Указатель <i>this</i>	272
11.1.6. Setter и getter	274
11.1.7. Свойство <i>__proto__</i>	276
11.1.8. Атрибуты свойств и метод <i>Object.create()</i>	278
11.1.9. Получение значений атрибутов свойств	280
11.1.10. Методы <i>defineProperty()</i> и <i>defineProperties()</i>	282
11.1.11. Ограничение доступа к объекту.....	284
11.1.12. Получение списка имен свойств	287
11.1.13. Перебор свойств объекта.....	289
11.1.14. Проверка существования свойств и методов.....	290
11.1.15. Свойства с типом <i>symbol</i>	291
11.1.16. Сравнение объектов	292
11.1.17. Создание копии объекта.....	293
11.1.18. Объединение объектов.....	295
11.1.19. Удаление свойства	296
11.2. Создание объекта с помощью класса <i>Object</i>	298
11.3. Создание класса (старый стиль).....	299
11.3.1. Функция в качестве конструктора класса	300
11.3.2. Прототипы	301
11.3.3. Свойство <i>constructor</i>	303
11.3.4. Наследование.....	303
11.3.5. Переопределение методов	305
11.3.6. Статические свойства и методы.....	307
11.4. Создание класса (новый стиль)	307
11.4.1. Инструкция <i>class</i>	308
11.4.2. Конструктор класса.....	308
11.4.3. Наследование.....	309
11.4.4. Переопределение методов	310
11.4.5. Публичные свойства	311
11.4.6. Приватные свойства и методы	312
11.4.7. Статические свойства и методы.....	313
11.5. Оператор <i>instanceof</i>	315
11.6. Массивоподобные объекты (псевдомассивы).....	316
11.7. Итераторы	317
11.8. Методы-генераторы.....	319
11.9. Пространства имен	320
11.10. Класс <i>Proxy</i>	322
11.10.1. Создание проху-объекта.....	322
11.10.2. Ограничение доступа к свойствам объекта.....	324
11.10.3. Ограничение проверки существования свойства	327
11.10.4. Ограничение получения списка свойств и их атрибутов	328
11.10.5. Ограничение добавления свойства и изменения атрибутов	330

11.10.6. Ограничение удаления свойства	331
11.10.7. Ограничение нерасширяемости объекта	332
11.10.8. Ограничение доступа к прототипу	333
11.10.9. Перехват создания объекта	334
11.10.10. Перехват вызова функции	335
11.10.11. Отключаемый проху-объект	335
11.11. Класс <i>Reflect</i>	336
11.12. Формат JSON: преобразование объекта в строку и обратно	340

Глава 12. Модули.....345

12.1. Модули CommonJS	345
12.1.1. Подключение встроенных модулей	345
12.1.2. Получение списка встроенных модулей	346
12.1.3. Подключение пользовательских модулей	347
12.1.4. Кэширование модулей	350
12.1.5. Подключение каталогов	352
12.1.6. Подключение модулей и пакетов из каталога <i>node_modules</i>	353
12.1.7. Переменная окружения <i>NODE_PATH</i>	354
12.1.8. Экспорт идентификаторов из модуля	355
12.1.9. Объект <i>module</i>	357
12.2. Модули ECMAScript (ESM)	358
12.2.1. Основные отличия модулей ECMAScript от модулей CommonJS	358
12.2.2. Подключение встроенных модулей	359
12.2.3. Получение списка встроенных модулей	360
12.2.4. Подключение пользовательских модулей	361
12.2.5. Кэширование модулей	363
12.2.6. Подключение модулей и пакетов из каталога <i>node_modules</i>	364
12.2.7. Экспорт идентификаторов из модуля	364
12.2.8. Экспорт идентификатора по умолчанию	367
12.2.9. Реэкспорт	369
12.2.10. Динамический импорт	371
12.2.11. Подключение модулей CommonJS	373
12.2.12. Подключение модуля ECMAScript внутри модуля CommonJS	374
12.2.13. Подключение модулей в формате <i>JSON</i>	375
12.3. Свойства <i>main</i> , <i>exports</i> и <i>imports</i> в файле <i>package.json</i>	376
12.4. Переменные уровня приложения	385

Глава 13. Обработка ошибок.....387

13.1. Типы ошибок	387
13.2. Исключения и их обработка. Инструкция <i>try...catch...finally</i>	389
13.3. Класс <i>Error</i> : объекты исключения	392
13.4. Оператор <i>throw</i> : генерирование исключений	393
13.5. Поиск ошибок в программе	394
13.6. Метод <i>assert()</i>	396
13.7. Отладка программы в редакторе VS Code	396
13.8. Строгий режим	402
13.9. Установка и настройка <i>Prettier</i>	407
13.10. Установка и настройка <i>ESLint</i>	415

Глава 14. Асинхронность	421
14.1. Объект класса <i>Promise</i>	421
14.1.1. Создание объекта класса <i>Promise</i>	421
14.1.2. Обработка изменения статуса	423
14.1.3. Массовая обработка объектов класса <i>Promise</i>	425
14.2. Ключевые слова <i>async</i> и <i>await</i>	429
14.3. Цикл <i>for await...of</i>	431
14.4. Асинхронные итераторы.....	433
14.5. Асинхронные методы-генераторы	434
14.6. Обработка событий.....	435
14.6.1. Генерирование событий.....	436
14.6.2. Назначение обработчиков событий	437
14.6.3. Удаление обработчиков событий.....	440
14.6.4. Асинхронная обработка событий.....	441
Глава 15. Класс <i>Buffer</i>: массив байтов фиксированного размера	443
15.1. Создание массива байтов	443
15.2. Определение размера массива.....	447
15.3. Получение и изменение значения по индексу	448
15.4. Запись и чтение данных	448
15.5. Создание копии массива	451
15.6. Получение части массива.....	452
15.7. Объединение массивов.....	453
15.8. Изменение порядка следования байтов	453
15.9. Перебор элементов массива.....	454
15.10. Методы, возвращающие итераторы.....	455
15.11. Сравнение массивов	456
15.12. Проверка наличия значения в массиве	457
15.13. Преобразование массива в строку или в другой объект.....	459
15.14. Преобразование кодировок.....	460
Глава 16. Чтение и запись файлов	462
16.1. Указание пути к файлу или каталогу	462
16.2. Модуль <i>path</i> : преобразование путей	466
16.3. Запись в файл с указанием пути к файлу	470
16.4. Чтение из файла с указанием пути к файлу.....	480
16.5. Открытие и закрытие файла	485
16.6. Режимы открытия файла.....	489
16.7. Запись в файл с указанием дескриптора.....	491
16.8. Чтение из файла с указанием дескриптора	500
16.9. Изменение размера файла.....	510
16.10. Дескрипторы стандартных потоков ввода/вывода	514
Глава 17. Файловые потоки ввода/вывода	516
17.1. Класс <i>WriteStream</i> : поток вывода в файл.....	516
17.1.1. Создание потока вывода.....	516
17.1.2. Запись данных	518
17.1.3. События потока вывода.....	519

17.2. Класс <i>ReadStream</i> : поток ввода из файла.....	520
17.2.1. Создание потока ввода.....	520
17.2.2. Чтение данных.....	521
17.2.3. События потока ввода.....	522
17.2.4. Метод <i>pipe()</i>	523
17.2.5. Чтение файла, сохраненного в русской кодировке.....	524
17.3. Перенаправление стандартных потоков ввода/вывода.....	524
Глава 18. Работа с файловой системой.....	526
18.1. Переименование и перемещение файлов.....	526
18.2. Копирование файлов.....	529
18.3. Удаление файлов.....	532
18.4. Получение сведений о файлах и каталогах.....	535
18.5. Права доступа к файлу и каталогу.....	542
18.6. Проверка существования файлов и каталогов.....	544
18.7. Создание каталогов.....	545
18.8. Создание временных каталогов.....	547
18.9. Удаление каталогов.....	548
18.10. Перебор элементов в каталоге.....	551
18.11. Отслеживание изменения файла или каталога.....	556
Глава 19. Веб-сервер на Node.js.....	560
19.1. Создание и запуск веб-сервера.....	560
19.2. Объект запроса.....	562
19.3. Объект ответа.....	563
19.4. Отправка файла в качестве серверного ответа.....	565
19.5. <i>Nodemon</i> : автоматическая перезагрузка веб-сервера.....	566
Глава 20. Работа JavaScript в веб-браузерах.....	568
20.1. Особенности работы JavaScript в веб-браузерах.....	568
20.2. Первая JavaScript-программа.....	569
20.3. Тег <i><script></i>	570
20.4. Расположение JavaScript-программы.....	571
20.5. Расположение определений функций в HTML-коде.....	574
20.6. Консоль Mozilla Firefox.....	575
20.7. Встроенные диалоговые окна.....	576
20.7.1. Окно с сообщением и кнопкой <i>ОК</i>	576
20.7.2. Окно с сообщением и кнопками <i>ОК</i> и <i>Отмена</i>	577
20.7.3. Окно с полем ввода и кнопками <i>ОК</i> и <i>Отмена</i>	577
20.8. Получение доступа к элементам веб-страницы.....	578
20.9. Создание часов на веб-странице.....	579
20.10. Отладчик Mozilla Firefox.....	580
20.11. JavaScript-библиотеки.....	582
Глава 21. Объектная модель документа.....	584
21.1. Структура объектной модели документа.....	584
21.2. Объект <i>window</i>	585
21.3. Работа с фреймами.....	586
21.4. Объект <i>navigator</i> : получение сведений о веб-браузере.....	586

21.5. Объект <i>screen</i> : получение сведений об экране клиентского компьютера.....	587
21.6. Объект <i>location</i> : доступ к интернет-адресу веб-страницы	587
21.7. Объект <i>history</i> : история веб-браузера	588
21.8. Объект <i>document</i> : работа с веб-страницей.....	588
21.9. Узлы DOM.....	591
21.10. Общие свойства и методы элементов веб-страницы	595
21.11. Работа с таблицами стилей	596
21.12. Объект <i>selection</i> : работа с выделением	599
21.13. Объект <i>Range</i> : работа с фрагментами текста	601
21.14. Cookie: хранение данных на компьютере клиента.....	604
21.15. Хранилище	607
21.15.1. Сессионное и локальное хранилища	608
21.15.2. Работа с хранилищем	608
21.15.3. Использование локального хранилища для хранения данных	609
21.16. Работа с графическими изображениями	610
21.17. Работа с мультимедиа.....	611
21.18. Средства геолокации	613
21.18.1. Доступ к средствам геолокации	613
21.18.2. Получение данных геолокации	613
21.18.3. Обработка нештатных ситуаций	614
21.18.4. Задание дополнительных параметров	615
21.18.5. Отслеживание местоположения компьютера	616
Глава 22. События	617
22.1. Назначение обработчиков событий	617
22.2. Удаление обработчиков событий.....	619
22.3. Указатель <i>this</i>	619
22.4. Объект <i>event</i>	620
22.5. Действия по умолчанию и их отмена.....	621
22.6. Всплытие событий.....	623
22.7. Фазы событий	625
22.8. События веб-страницы	626
22.9. События мыши.....	627
22.10. События клавиатуры	630
22.11. События аудио- и видеопроигрывателей.....	631
Глава 23. Взаимодействие с элементами форм	633
23.1. Элементы формы	633
23.2. Коллекция <i>forms</i> . Доступ к элементу формы из скрипта	634
23.3. Работа с формами	635
23.4. Работа с элементами формы	636
23.4.1. Текстовые поля, поля ввода пароля, подстроки поиска, адреса электронной почты, интернет-адреса и телефона.....	638
23.4.2. Поле ввода числа и регулятор	640
23.4.3. Поля ввода даты и времени	641
23.4.4. Поле выбора цвета	641
23.4.5. Поле выбора файла	641
23.4.6. Область редактирования.....	642
23.4.7. Флажок и переключатель	644

23.4.8. Список	645
23.4.9. Кнопки	649
23.5. Расширенная проверка значения, занесенного в поле ввода	651
Глава 24. Холст: программируемая графика	653
24.1. Тег <code><canvas></code>	653
24.2. Создание контекста рисования	653
24.3. Заливка	654
24.4. Контур	654
24.5. Рисование прямоугольников	656
24.6. Очистка области холста	656
24.7. Вывод текста	657
24.8. Вывод графических изображений	658
24.9. Рисование сложных фигур	659
24.10. Определение вхождения точки в состав контура	662
24.11. Задание сложных цветов	662
24.11.1. Линейный градиент	662
24.11.2. Радиальный градиент	663
24.11.3. Заливка текстурой	664
24.12. Сохранение и восстановление состояния	664
24.13. Преобразования	665
24.14. Управление наложением графики	665
24.15. Задание уровня прозрачности	666
24.16. Создание тени	667
24.17. Работа с отдельными пикселями	667
24.17.1. Получение массива пикселей	667
24.17.2. Создание пустого массива пикселей	668
24.17.3. Манипуляция пикселями	668
24.17.4. Вывод массива пикселей	669
Глава 25. AJAX: обмен данными без перезагрузки веб-страницы	671
25.1. Основы технологии AJAX	671
25.1.1. Обмен данными с помощью тега <code><iframe></code>	671
25.1.2. Объект <code>XMLHttpRequest</code>	675
25.1.3. Получение данных в текстовом формате	680
25.1.4. Получение данных в формате XML	685
25.1.5. Получение данных в формате JSON	692
25.2. <i>Fetch API</i>	698
25.2.1. Функция <code>fetch()</code>	698
25.2.2. Классы <code>URL</code> и <code>URLSearchParams</code>	701
25.2.3. Объект запроса <code>Request</code>	707
25.2.4. Отправка данных при выгрузке веб-страницы	711
25.2.5. Прерывание запроса	712
25.2.6. Класс <code>FormData</code>	712
25.2.7. Отправка файлов	715
25.2.8. Объект ответа <code>Response</code>	718
25.2.9. Класс <code>Headers</code>	721
25.2.10. Кросс-доменные запросы	724

Глава 26. Сборка веб-сайтов	730
26.1. Файл <i>.browserslistrc</i>	730
26.2. Сборка CSS- и SCSS-файлов.....	731
26.2.1. Добавление вендорных префиксов	731
26.2.2. Сжатие CSS-файлов	732
26.2.3. SCSS-транслятор <i>node-sass</i>	733
26.3. Пакет <i>npm-run-all</i> : запуск нескольких скриптов.....	736
26.4. Сборщик проектов Webpack	737
26.4.1. Создание файла конфигурации	737
26.4.2. Сборка JavaScript-файлов	738
26.4.3. Несколько точек входа.....	739
26.4.4. Очистка содержимого каталога	741
26.4.5. Подключение файлов в формате JSON	742
26.4.6. Подключение библиотек из каталога <i>node_modules</i>	742
26.4.7. Пакет <i>babel-loader</i>	743
26.4.8. Сборка SCSS-файлов	744
26.4.9. Подключение изображений	746
26.4.10. Копирование файлов	747
Заключение.....	749
Приложение. Описание электронного архива.....	750
Предметный указатель	751

Введение

JavaScript (также известен как *ECMAScript*) — язык программирования, созданный для написания скриптов — программ, которые встраиваются в веб-страницы и выполняются непосредственно веб-браузером. Самый простой скрипт может в ответ на возникновение какого-либо события (например, наведения мыши на определенный элемент страницы) выводить на странице заданное сообщение (скажем, поясняющий текст). Более сложный скрипт способен прочесть значения, занесенные пользователем в форму, чтобы проверить их на корректность перед отправкой на сервер или даже обработать самостоятельно и тут же вывести результат. И наконец, высший пилотаж JavaScript-программирования — программная подгрузка данных с сервера для формирования на их основе содержимого целой страницы.

JavaScript оказался настолько удачным языком, что остальные программисты стали завидовать разработчикам веб-страниц. И вот в 2009 году появилась первая версия программной платформы Node.js, которая была создана на основе интерпретатора (исполняющей среды) языка JavaScript, встроенного в популярный браузер Google Chrome, и позволяла исполнять JavaScript-программы без веб-браузера. С появлением Node.js появилась возможность писать на JavaScript программы практически любого назначения: инструментальные и системные утилиты, серверы и даже приложения с графическим интерфейсом (правда, для этого понадобятся дополнительные библиотеки).

Предлагаемая вниманию читателей книга рассказывает о языке JavaScript, его применении для написания как программ, работающих под управлением Node.js, так и скриптов, выполняемых веб-браузерами. По сути, это всеобъемлющее руководство для JavaScript-программистов, желающих знать об этом языке все.

Структура книги

- В *главе 1* описывается редактор программного кода Visual Studio Code, который предназначен именно для программистов и которым мы будем пользоваться на протяжении всего обучения. Редактор корректно работает с кодировкой UTF-8, имеет подсветку синтаксиса языков HTML, CSS, JavaScript и др., выводит различные полезные подсказки и, вообще, всячески помогает разработчику.

- ❑ *Глава 2* является вводной. Мы установим платформу Node.js, создадим и запустим в ней первую программу — из командной строки и из редактора Visual Studio Code. Кроме того, вкратце разберемся со структурой программы, научимся выводить результаты ее работы и получать данные от пользователя.
- ❑ В *главе 3* мы познакомимся с переменными и типами данных, поддерживаемыми JavaScript.
- ❑ В *главе 4* мы рассмотрим операторы, выполняющие различные действия с данными, в частности изучим операторы ветвления и циклы, позволяющие изменять порядок выполнения инструкций в программе.
- ❑ *Глава 5* полностью посвящена работе с числами. Мы узнаем, какие числовые типы данных поддерживает JavaScript, научимся применять различные функции, генерировать случайные числа и др.
- ❑ *Глава 6* познакомит с массивами и множествами JavaScript. Мы научимся создавать массивы, одно- и многомерные, перебирать элементы массива, сортировать, выполнять поиск значений в массивах и др.
- ❑ *Глава 7* полностью посвящена работе со строками.
- ❑ В *главе 8* мы рассмотрим регулярные выражения, которые очень пригодятся при выполнении сложного поиска или замены в строках.
- ❑ *Глава 9* рассказывает о работе со значениями даты и времени (их еще называют временными метками).
- ❑ В *главе 10* мы научимся создавать пользовательские функции — фрагменты произвольного программного кода, которые можно вызывать на выполнение в любом месте программы.
- ❑ *Глава 11* познакомит нас с *объектно-ориентированным программированием (ООП)* — еще одним способом многократного использования кода. В отличие от функций, ООП позволяет описать предметы реального мира в виде объектов и организовать связи между этими объектами.
- ❑ В *главе 12* мы научимся размещать код в отдельных файлах — *модулях*, выполнять экспорт идентификаторов из одного модуля и импортировать эти идентификаторы для использования в других модулях.
- ❑ В *главе 13* мы рассмотрим способы поиска ошибок в программе и научимся отлаживать код в отладчике, встроенном в Visual Studio Code.
- ❑ *Глава 14* познакомит нас с асинхронным выполнением программного кода и событиями Node.js.
- ❑ В *главе 15* мы рассмотрим класс `Buffer`, который позволит работать с отдельными байтами и выполнять преобразование кодировок.
- ❑ *Главы 16, 17 и 18* научат работать с файлами и каталогами, читать и записывать файлы в различных форматах.
- ❑ В *главе 19* мы напишем и запустим собственный веб-сервер.

- ❑ В *главе 20* мы оставим в покое Node.js и познакомимся с особенностями исполнения JavaScript-скриптов в веб-браузерах.
- ❑ *Глава 21* рассматривает объектную модель документа (DOM) веб-браузера, благодаря которой мы сможем получить из скрипта доступ к странице и ее элементам и управлять ими: менять содержимое элементов, добавлять, изменять и удалять элементы страницы, работать со стилями CSS и др.
- ❑ *Глава 22* познакомит нас с многочисленными событиями, возникающими в веб-страницах и их элементах, и обработкой этих событий.
- ❑ В *главе 23* мы научимся программно обрабатывать данные, занесенные пользователем в элементы формы.
- ❑ *Глава 24* познакомит нас с программным рисованием на холсте, реализуемым тегом `<canvas>`.
- ❑ В *главе 25* рассматривается технология AJAX, предназначенная для фоновой загрузки произвольных данных с сервера, и два реализующих ее программных инструмента.
- ❑ И наконец, в *главе 26* мы рассмотрим готовые программные пакеты для Node.js, которые могут помочь при разработке веб-сайтов. Например, пакет sass транслирует таблицы стилей, написанные на языках SCSS и SASS, в поддерживаемый браузерами CSS. А пакет Webpack может оказаться очень полезным при сборке сложных веб-проектов из разнородных исходных файлов.

Для полного понимания материала книги от читателя потребуются знания HTML и CSS в объеме первых двух глав книги "HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера".

Все листинги из этой книги вы найдете в файле Listings.doc, электронный архив с которым можно загрузить с FTP-сервера издательства "БХВ" по ссылке: <ftp://ftp.bhv.ru/9785977568470.zip> или со страницы книги на сайте <https://bhv.ru> (см. приложение).

Типографские соглашения

В книге будут часто приводиться различные языковые конструкции, применяемые в программировании. Для наглядности при их написании приняты следующие типографские соглашения (в реальном коде они недействительны):

- ❑ HTML-, CSS-, JavaScript- и PHP-код набран моноширинным шрифтом:

```
for (var i = 0; i < 5; i++) {  
    console.log(i);  
}  
console.log('i =', i);
```

- ❑ в угловые скобки `<>` заключены наименования различных языковых конструкций. В код программы, разумеется, должны быть подставлены реальные конструкции. Например:

```
const <Имя константы> = <Значение константы>
```

Здесь вместо фраз <Имя константы> и <Значение константы> должны быть подставлены реальные имя и значение;

- в квадратные скобки [] заключены необязательные фрагменты кода:

```
process.exit([<Код завершения>]);
```

Здесь параметр <Код завершения> может указываться, а может и отсутствовать;

- вертикальной чертой | разделены доступные для выбора языковые конструкции, из которых в код можно подставить лишь одну:

```
var|let <Переменная>
```

Здесь можно подставить либо ключевое слово var, либо слово let;

- слишком длинные, не помещающиеся на одной строке книги фрагменты кода разделены на несколько строк, и в местах разрывов поставлены знаки ↵:

```
node-sass -o dist/css/ --include-path node_modules/ ↵
scss/main.scss
```

Приведенный здесь код хотя и разбит на две строки, но должен быть набран в одну. Символ ↵ при этом следует удалить;

- троеточием . . . помечены фрагменты кода, пропущенные ради сокращения объема текста книги:

```
class SomeClass {
    *#method1() { . . . }
    static *staticMethod2() { . . . }
    static *#staticMethod3() { . . . }
}
```

Здесь весь код внутри фигурных скобок пропущен.

НЕ ЗАБЫВАЙТЕ!

Все приведенные здесь типографские соглашения имеют смысл только в примерах написания языковых конструкций. В реальном программном коде они приведут к возникновению ошибки.

Желаем приятного изучения и надеемся, что книга поможет вам написать сначала простые, а потом и более сложные веб-приложения.



ГЛАВА 1

Редактор Visual Studio Code

Для создания файлов с кодом из книги можно воспользоваться любым текстовым редактором, например Блокнотом. Однако мы будем работать с многобайтовой кодировкой UTF-8, а Блокнот при сохранении в этой кодировке добавляет *метку порядка байтов* (сокращенно *BOM*), которая может стать причиной программных ошибок. Чтобы избежать этого, в качестве редактора кода на протяжении всего обучения воспользуемся программой *Visual Studio Code* (далее *VS Code*). Она также позволяет корректно работать с однобайтовой кодировкой Windows-1251 и имеет подсветку синтаксиса языков HTML, CSS, JavaScript и др.

Для быстрого редактирования файлов советуем дополнительно установить редактор Notepad++. Скачать Notepad++ можно абсолютно бесплатно со страницы <https://notepad-plus-plus.org/>. Из двух вариантов (архив и инсталлятор) советуем выбрать именно инсталлятор, т. к. при установке можно будет указать язык интерфейса программы. Установка Notepad++ предельно проста и в пояснениях не нуждается.

1.1. Установка VS Code

Для установки VS Code переходим на сайт <https://code.visualstudio.com/>, прокручиваем страницу в самый низ, в раздел загрузок, и скачиваем установщик **User Installer 64 bit** (для 64-разрядной редакции Windows) или **User Installer 32 bit** (для ее 32-разрядной редакции). Файл установщика будет иметь имя VSCODEUSERSETUP-X64-1.55.2.EXE или VSCodeUserSetup-ia32-1.55.2.exe соответственно.

Запускаем программу установки. На первом шаге принимаем лицензионное соглашение и нажимаем кнопку **Далее** (рис. 1.1). На втором шаге при желании можно изменить каталог, в который будет установлена программа, или оставить каталог по умолчанию (рис. 1.2). Нажимаем кнопку **Далее**. На следующем шаге имеется возможность изменить имя группы, которая будет создана в меню **Пуск** после установки программы, также можно оставить имя группы по умолчанию (рис. 1.3). Нажимаем кнопку **Далее**. На следующем шаге устанавливаем все флажки, кроме **Зарегистрировать Code в качестве редактора для поддерживаемых типов файлов** (рис. 1.4). Для этой цели лучше использовать более простой редактор,

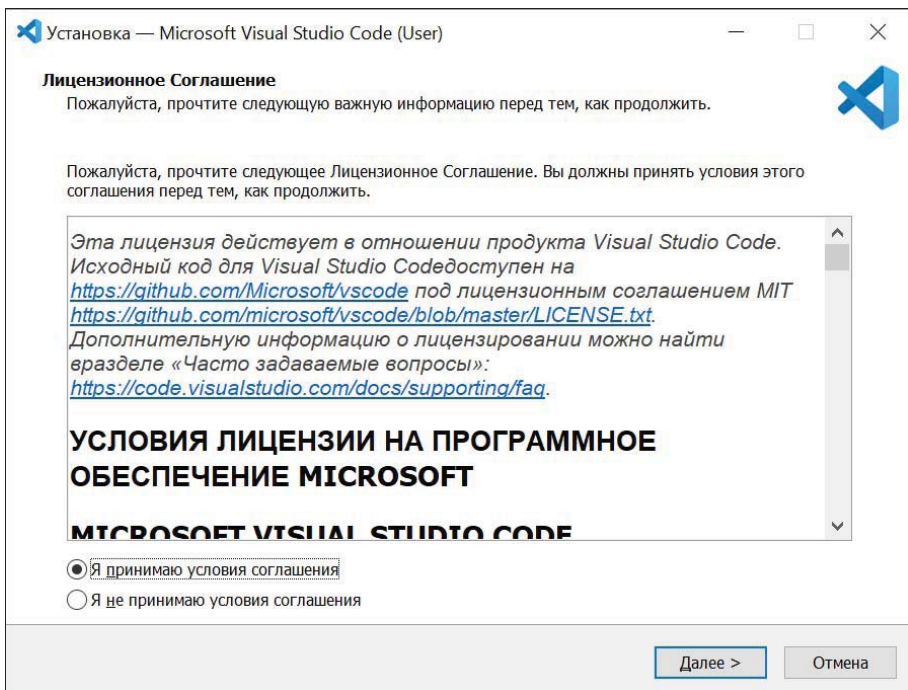


Рис. 1.1. Установка VS Code: шаг 1

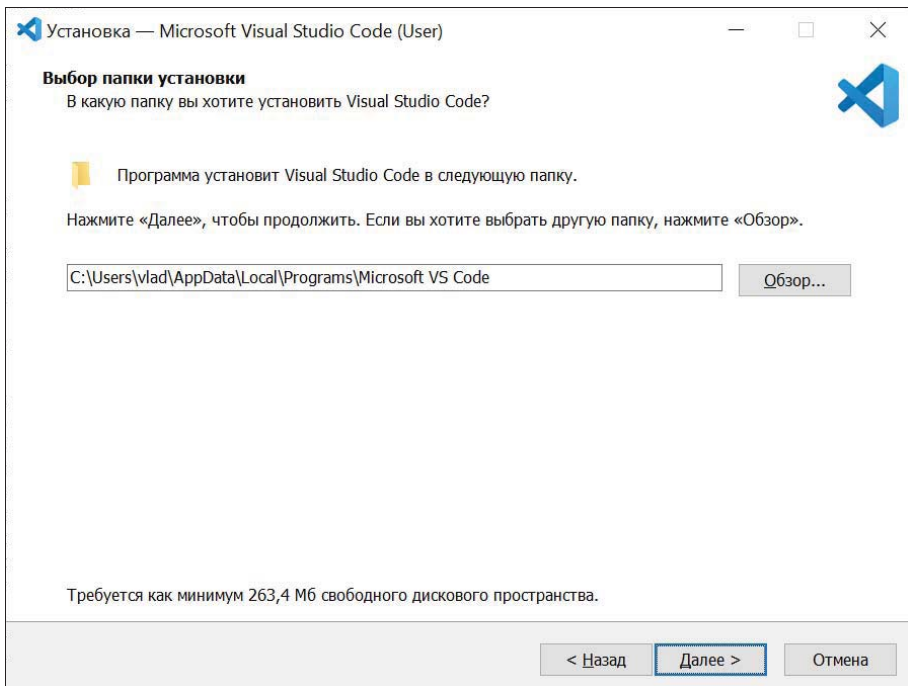


Рис. 1.2. Установка VS Code: шаг 2

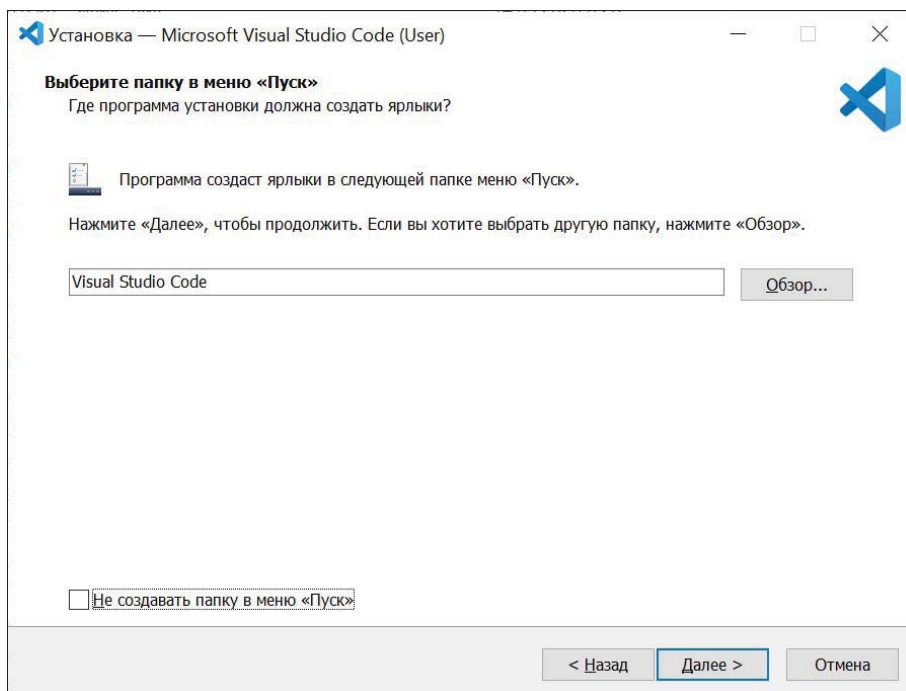


Рис. 1.3. Установка VS Code: шаг 3

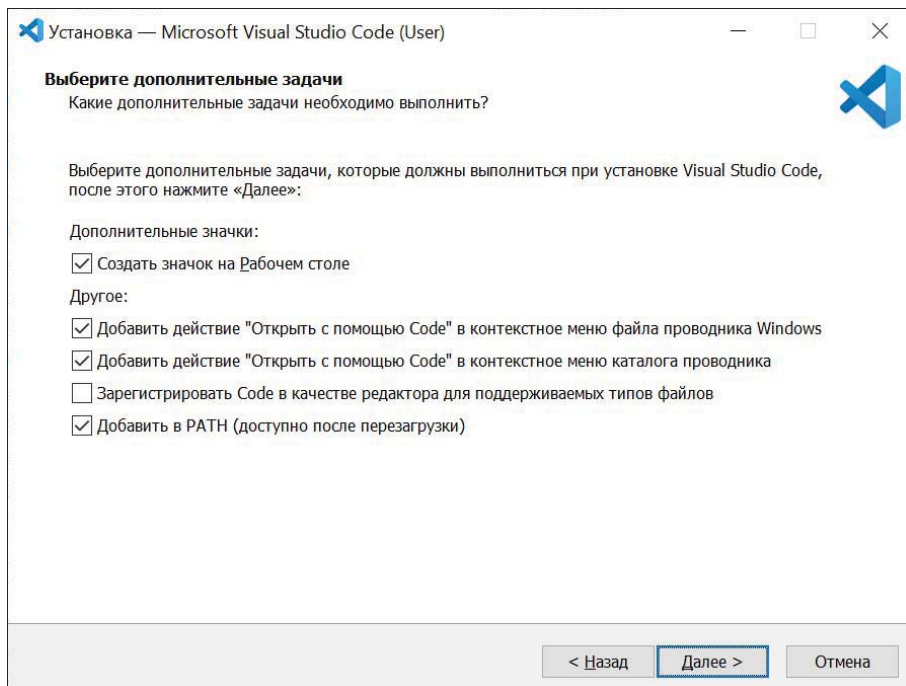


Рис. 1.4. Установка VS Code: шаг 4

наподобие Notepad++. Нажимаем кнопку **Далее**. На очередном шаге проверяем настройки и нажимаем кнопку **Установить** (рис. 1.5) для запуска процесса установки. На последнем шаге, по окончании установки, нажимаем кнопку **Завершить** (рис. 1.6).

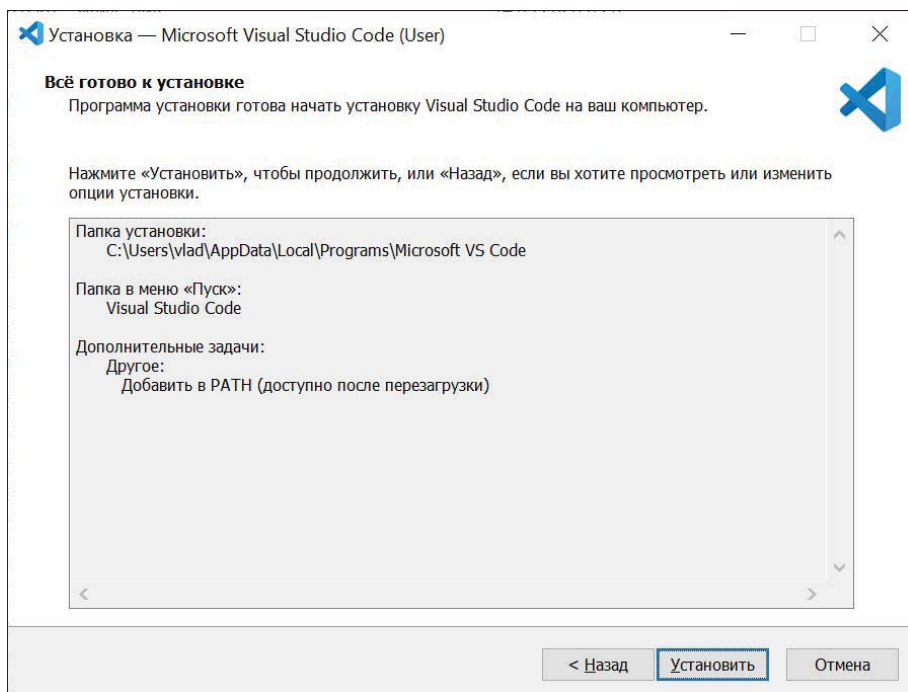


Рис. 1.5. Установка VS Code: шаг 5

По умолчанию, если на шаге 2 (см. рис. 1.2) не был выбран другой путь, редактор устанавливается в каталог `C:\Users\<Имя пользователя>\AppData\Local\Programs\Microsoft VS Code`. Путь до вложенного каталога `bin` записывается в переменную окружения `PATH` для текущего пользователя. Кроме того, на Рабочем столе создается ярлык для запуска редактора. В меню **Пуск** добавляется группа **Visual Studio Code** (имя по умолчанию, может быть изменено на шаге 3, см. рис. 1.3) с одноименным пунктом. В контекстное меню Проводника Windows добавляются пункты для открытия в редакторе файла или целого каталога. Чтобы открыть файл, щелкаем правой кнопкой мыши на значке файла и из контекстного меню выбираем пункт **Открыть с помощью Code**. Аналогичный пункт будет при щелчке правой кнопкой мыши на значке каталога или внутри свободной области каталога в Проводнике Windows.

Если на последнем шаге был установлен флажок **Запустить Visual Studio Code** (см. рис. 1.6), то редактор автоматически запустится. Если после запуска отобразилось черное окно, и долго ничего не меняется, нужно выполнить дополнительную настройку. Закрываем редактор. На Рабочем столе находим ярлык для запуска редактора и щелкаем на нем правой кнопкой мыши. Из контекстного меню выбираем

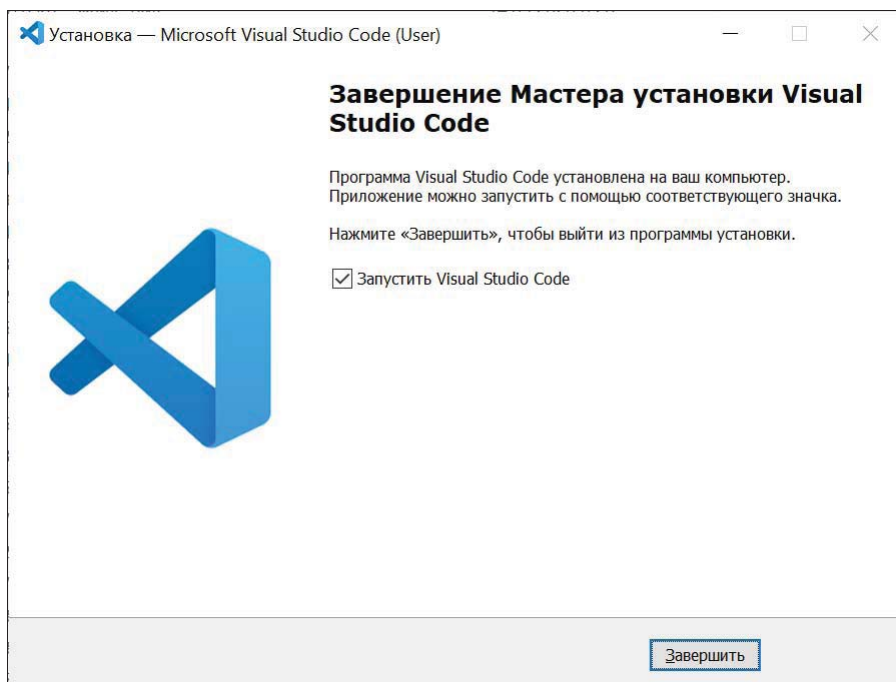


Рис. 1.6. Установка VS Code: шаг 6

пункт **Свойства**. В открывшемся окне на вкладке **Ярлык** в конец содержимого поля **Объект** добавляем пробел и флаг `--disable-gpu`:

```
"C:\Users\<Имя пользователя>\AppData\Local\Programs\Microsoft VS Code\Code.exe" --disable-gpu
```

Вместо `<Имя пользователя>` должно быть указано имя пользователя вашего компьютера. Сохраняем изменения и запускаем редактор двойным щелчком на ярлыке. Если редактор успешно запустился, то мы можем продолжить изучение, но предварительно произведем его настройку.

1.2. Настройка редактора

По умолчанию редактор не поддерживает русский язык. Добавить поддержку русского языка можно, установив расширение Russian Language Pack for Visual Studio Code. Для этого в меню **View** выбираем пункт **Extensions** или нажимаем комбинацию клавиш `<Shift>+<Ctrl>+<X>`. В результате в левой части окна редактора отобразится панель **Extensions**. В строке поиска вводим имя искомого расширения: "Russian Language Pack for Visual Studio Code". Как только найденное расширение появится в списке, устанавливаем его, нажав небольшую кнопку **Install**, которая находится в правом нижнем углу пункта списка, представляющего это расширение. После установки расширения перезапускаем редактор.

Дополнительно устанавливаем расширение Auto Complete Tag, которое автоматически установит еще два расширения: Auto Rename Tag и Auto Close Tag. Эти рас-

ширения упростят работу с HTML-тегами (например, при переименовании открывающего тега будет автоматически изменяться содержимое закрывающего тега). Поскольку редактор теперь "говорит" по-русски, для вывода панели **Расширения** следует выбрать одноименный пункт меню **Вид**, а для установки расширения — щелкнуть кнопку **Установить** в пункте списка.

ВНИМАНИЕ!

В дальнейшем будет описываться редакция Visual Studio Code, русифицированная с помощью расширения Russian Language Pack for Visual Studio Code.

НА ЗАМЕТКУ

Расширения сохраняются в каталоге `C:\Users\<Имя пользователя>\vscode\extensions`.

Теперь переходим к настройкам. В меню **Файл** выбираем пункт **Настройки | Параметры** или нажимаем комбинацию клавиш `<Ctrl>+<запятая>`. Выполняем настройку следующих параметров в разделе **Пользователь** (вводим их названия в поле для поиска):

- ☐ **Editor: Font Family** — названия используемых шрифтов через запятую. Рекомендуем значение "Consolas, 'Courier New', monospace".
- ☐ **Editor: Font Size** — размер шрифта. Можете подобрать удобное значение, например 16 пунктов.
- ☐ **Editor: Tab Size** — число пробелов в табуляции. Вводим значение 3.
- ☐ **Editor: Detect Indentation** — сбрасываем флажок, чтобы всегда использовать пробелы вместо символов табуляции.
- ☐ **Editor: Mouse Wheel Zoom** — если флажок установлен, изменять масштаб можно с помощью колесика мыши, удерживая нажатой клавишу `<Ctrl>`.
- ☐ **Editor: Word Wrap** — стиль переноса длинных строк. Из списка выбираем пункт **bounded**, включающий автоматический перенос длинных строк.
- ☐ **Editor: Word Wrap Column** — количество символов в строке. Должно быть указано значение 80 (задано по умолчанию).
- ☐ **Files: Auto Save** — управляет автоматическим сохранением файла. Рекомендуем из списка выбрать пункт **onFocusChange**, при котором файл будет автоматически сохраняться при потере фокуса ввода.
- ☐ **Editor: Links** — снимите флажок, если не хотите, чтобы внутри файла выделялись ссылки. С одной стороны, удобно открывать файл переходом по ссылке, с другой — подчеркнутые ссылки мешают.
- ☐ **Files: Encoding** — из списка выбираем пункт **utf8**, который обозначает кодировку UTF-8 без BOM (метки порядка байтов).
- ☐ **Files: Eol** — задает символ перевода строк. Из списка выбираем пункт `\n`. Чтобы сменить символ перевода строк для текущего файла, в строке состояния щелкаем на значке **CRLF** или **LF**, а затем из списка выбираем нужный пункт.

Точно таким же способом можно задать и другие настройки.

Все изменения настроек в разделе **Пользователь** сохраняются в файле settings.json, который расположен в каталоге C:\Users\<Имя пользователя>\AppData\Roaming\Code\User. Этот файл можно открыть в VS Code, щелкнув расположенную в правой части строки заголовков вкладок кнопку **Открыть параметры**, и изменить вручную.

Изменения настроек в разделе **Рабочая область** записываются в файле settings.json, созданном в каталоге .vscode, вложенном в каталог проекта. Таким образом, можно указывать какие-либо настройки на уровне отдельного проекта.

1.3. Смена цветовой темы, тем значков файлов и продукта

По умолчанию используется темная тема оформления. Для смены темы в меню **Файл** выбираем пункт **Настройки | Цветовая тема** или нажимаем комбинацию клавиш <Ctrl>+<K>, а затем комбинацию <Ctrl>+<T>. В итоге отобразится список установленных тем (рис. 1.7). Доступны темы светлые, темные и с высокой контрастностью. Существует также возможность установить тему как расширение или создать свою собственную тему. При выборе пункта из списка внешний вид редактора сразу изменится. Советуем попробовать выбрать темную тему Monokai.

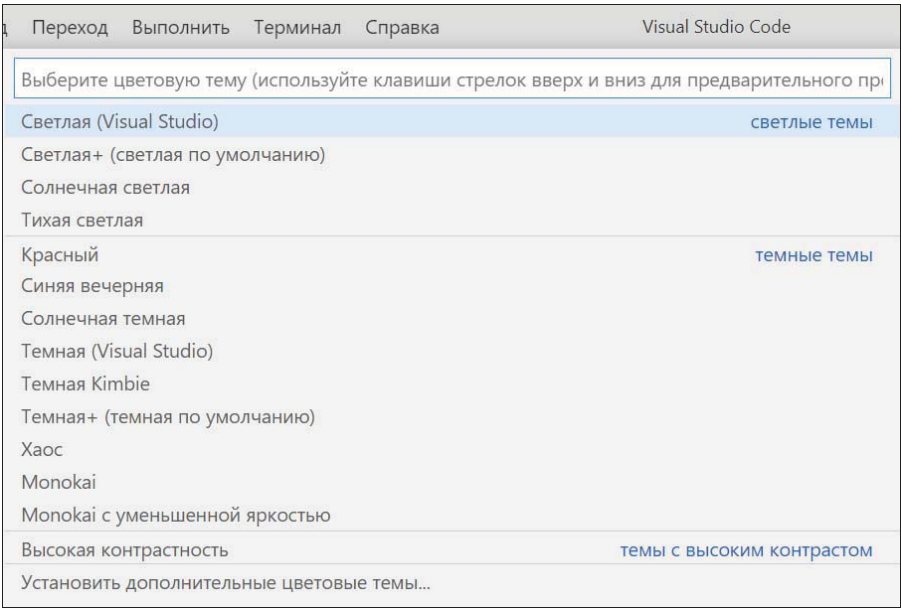


Рис. 1.7. Выбор цветовой темы

Все установленные темы доступны для выбора в настройках редактора. В строке поиска настроек введите **Workbench: Color Theme**, затем выберите тему из раскрывающегося списка (рис. 1.8). Так, при выборе темы "Светлая (Visual Studio)" в файл settings.json будет добавлена следующая строка:

```
"workbench.colorTheme": "Visual Studio Light"
```

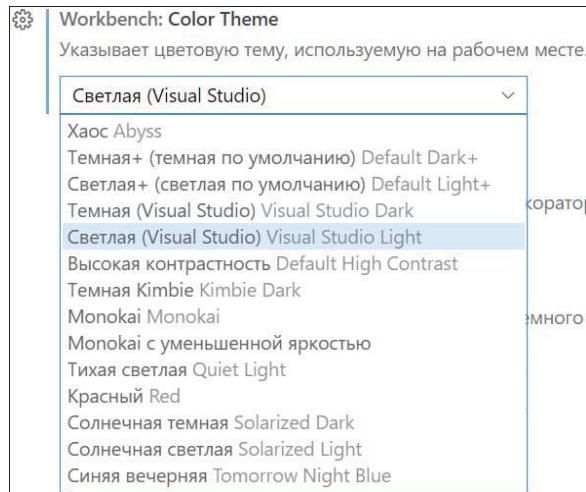



Рис. 1.8. Выбор цветовой темы в настройках редактора

Также имеется возможность изменить значки файлов, выводимые в панели **Проводник** (будет описана далее), задав другую тему значков. Для этого в меню **Файл** выбираем пункт **Настройки | Тема значков файлов**. На экране отобразится список доступных тем значков файлов, аналогичный списку цветовых тем (см. рис. 1.7), в котором можно выбрать нужную тему.

Тему значков файлов можно указать и в настройках редактора. В строке поиска настроек введите **Workbench: Icon Theme** и выберите тему из раскрывающегося списка, аналогичного списку цветовых тем (см. рис. 1.8). Так, при выборе темы "Минимальная (Visual Studio Code)" в файл settings.json будет добавлена следующая строка:

```
"workbench.iconTheme": "vs-minimal"
```

Наконец, можно изменить вид значков, выводящихся на панели действий редактора (будет описана далее), выбрав другую тему. В меню **Файл** выбираем пункт **Настройки | Тема значков продукта**. На экране появится список доступных тем значков продукта, аналогичный списку цветовых тем (см. рис. 1.7), в котором можно выбрать нужную тему. Правда, в этом списке изначально присутствует лишь одна тема, используемая по умолчанию, так что дополнительные темы придется устанавливать в панели **Расширения**.

Тему значков продукта можно задать и в настройках редактора, выполнив поиск по фразе **Workbench: Product Icon Theme** и выбрав тему из раскрывающегося списка. Так, при выборе темы "Fluent Icons" в файл settings.json будет добавлена следующая строка:

```
"workbench.productIconTheme": "fluent-icons"
```

ВНИМАНИЕ!

В книге будут приведены скриншоты, сделанные с использованием цветовой темы "Светлая (Visual Studio)", стандартной темы значков файлов "Seti (Visual Studio Code)" и стандартной темы значков продукта.

1.4. Структура окна редактора

При запуске редактора в центральной части окна отображается вкладка с приветствием (рис. 1.9). На ней расположены ссылки, с помощью которых можно создать файл, открыть каталог или последний проект, получить справочную информацию и т. д. Доступны также советы по установке расширений и настройке редактора. В нижней части вкладки расположен флажок **Отображать страницу приветствия при запуске**. Нам эта вкладка не понадобится, поэтому советуем сбросить флажок. В файл `settings.json` будет добавлена следующая строка:

```
"workbench.startupEditor": "newUntitledFile"
```

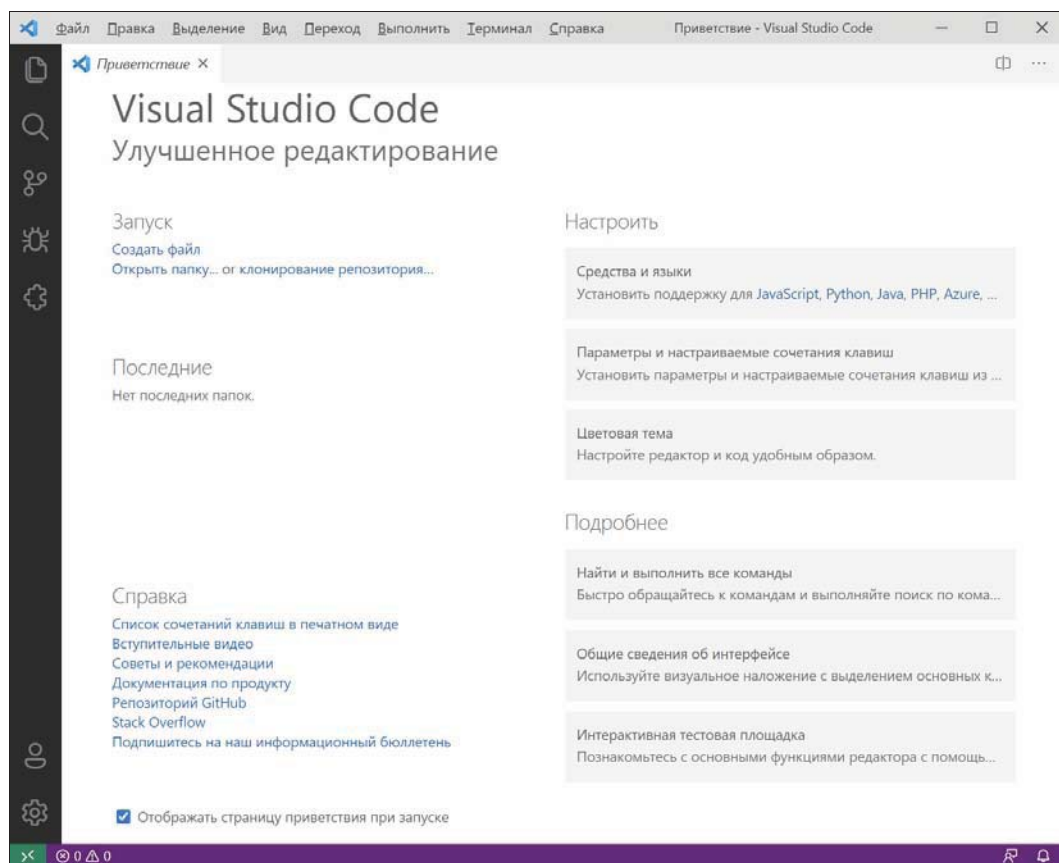


Рис. 1.9. Редактор Visual Studio Code

Значение `newUntitledFile` означает, что при запуске будет создаваться новый файл. Нас это также не устраивает. Переходим в настройки редактора и в строке поиска настроек вводим **Workbench: Startup Editor**. Из списка выбираем пункт **none**. В файл `settings.json` будет добавлена следующая строка вместо предыдущей:

```
"workbench.startupEditor": "none"
```


В результате при запуске редактор не будет содержать никаких вкладок. Чтобы вывести вкладку с приветствием, в меню **Справка** выбираем пункт **Приветствие** или удаляем из файла settings.json строку с упомянутой ранее настройкой.

1.4.1. Главное меню

В верхней части окна расположена строка меню, имя файла, открытого в активной вкладке, и название редактора, а также стандартные для окон кнопки **Свернуть**, **Развернуть** и **Заккрыть**. Кратко рассмотрим структуру главного меню.

❑ Меню **Файл** — команды для работы с файлами, каталогами и проектами (рис. 1.10). При выборе пункта **Настройки** отобразится меню с командами, позволяющими выполнить настройку редактора (рис. 1.11).

Файл	Правка	Выделение	Вид	Переход
Новый файл CTRL+N				
Новое окно CTRL+SHIFT+N				
Открыть файл... CTRL+O				
Открыть папку... CTRL+K CTRL+O				
Открыть рабочую область...				
Открыть последние >				
Добавить папку в рабочую область...				
Сохранить рабочую область как...				
Сохранить CTRL+S				
Сохранить как... CTRL+SHIFT+S				
Сохранить все CTRL+K S				
✓ Автосохранение				
Настройки >				
Отменить изменения в файле				
Заккрыть редактор CTRL+F4				
Заккрыть папку CTRL+K F				
Заккрыть окно CTRL+W				
Выход				

Рис. 1.10. Меню Файл

Параметры	CTRL+,
Параметры веб-служб	
Расширения	CTRL+SHIFT+X
Сочетания клавиш CTRL+K CTRL+S	
Раскладки клавиатуры	CTRL+K CTRL+M
Пользовательские фрагменты кода	
Цветовая тема	CTRL+K CTRL+T
Тема значков файлов	
Тема значков продукта	
Включение синхронизации параметров...	

Рис. 1.11. Меню Файл, пункт Настройки

- ❑ Меню **Правка** — команды для редактирования кода (рис. 1.12).
- ❑ Меню **Выделение** — команды для работы с выделением и курсорами (рис. 1.13).
- ❑ Меню **Вид** — команды для настройки внешнего вида редактора (рис. 1.14). При выборе пункта **Внешний вид** появится меню с командами для управления панелями (рис. 1.15), а при выборе пункта **Макет редактора** — меню с командами, позволяющими разделить содержимое окна редактора на несколько областей (рис. 1.16).

Правка	Выделение	Вид	Переход	Выполнить	Терминал
Отменить				CTRL+Z	
Повторить				CTRL+Y	
Вырезать				CTRL+X	
Копировать				CTRL+C	
Вставить				CTRL+V	
Найти				CTRL+F	
Заменить				CTRL+H	
Найти в файлах				CTRL+SHIFT+F	
Заменить в файлах				CTRL+SHIFT+H	
Переключить комментарий строки				CTRL+/	
Переключить комментарий блока				SHIFT+ALT+A	
Emmet: развернуть сокращение				Tab	

Рис. 1.12. Меню Правка

Выделение	Вид	Переход	Выполнить	Терминал	Справка	Untitled-1 - Visual Studio Code
Выделить все						CTRL+A
Развернуть выделение						SHIFT+ALT+RightArrow
Сжать выделение						SHIFT+ALT+LeftArrow
Копировать на строку выше						SHIFT+ALT+UpArrow
Копировать на строку ниже						SHIFT+ALT+DownArrow
Переместить на строку выше						ALT+UpArrow
Переместить на строку ниже						ALT+DownArrow
Дублировать выбранное						
Добавить курсор выше						CTRL+ALT+UpArrow
Добавить курсор ниже						CTRL+ALT+DownArrow
Добавить курсоры в окончания строк						SHIFT+ALT+I
Добавить следующее вхождение						CTRL+D
Добавить предыдущее вхождение						
Выбрать все вхождения						CTRL+SHIFT+L
Для работы в режиме нескольких курсоров нажмите левую кнопку мыши, удерживая клавишу CTRL						
Режим выбора столбцов						

Рис. 1.13. Меню Выделение

Вид	Переход	Выполнить	Терминал	Справ
Палитра команд...		CTRL+SHIFT+P		
Открыть представление...				
Внешний вид				>
Макет редактора				>
Проводник		CTRL+SHIFT+E		
Поиск				
Диспетчер служб		CTRL+SHIFT+G		
Выполнить		CTRL+SHIFT+D		
Расширения		CTRL+SHIFT+X		
Проблемы		CTRL+SHIFT+M		
Выходные данные		CTRL+SHIFT+U		
Консоль отладки		CTRL+SHIFT+Y		
Терминал		CTRL+`		
✓ Переносить по словам		ALT+Z		
✓ Показать мини-карту				
✓ Показать элементы навигации				
✓ Отображать пробелы				
Отобразить управляющие символы				

Рис. 1.14. Меню Вид

Полный экран	F11
Режим Zen	CTRL+K Z
Выровненный по центру макет	
✓ Показать строку меню	
Показать боковую панель	CTRL+B
✓ Показать строку состояния	
✓ Показать панель действий	
✓ Показать область редактора	
Показать панель	CTRL+J
Переместить боковую панель вправо	
Переместить панель влево	
Переместить панель вправо	
Увеличить	CTRL+=
Уменьшить	CTRL+-
Сбросить масштаб	CTRL+NumPad0

Рис. 1.15. Меню Вид, пункт Внешний вид

Разделить&	
Разделить вниз	
Разделить слева	
Разделить вправо	
Отдельный	
Два столбца	
Три столбца	
Две строки	
Три строки	
Сетка (2x2)	
Две строки вправо	
Два столбца внизу	
Отразить макет	SHIFT+ALT+0

Рис. 1.16. Меню Вид, пункт Макет редактора

Переход	Выполнить	Терминал	Справка	Untitled
Назад			ALT+LeftArrow	
Вперед			ALT+RightArrow	
Место последнего изменения			CTRL+K CTRL+Q	
Переключить редактор				>
Переключить группу				>
Перейти к файлу...			CTRL+P	
Перейти к символу в рабочей области...			CTRL+T	
Перейти к символу в редакторе...			CTRL+SHIFT+O	
Перейти к определению			F12	
Перейти к объявлению				
Перейти к определению типа				
Перейти к реализациям			CTRL+F12	
Перейти к ссылкам			SHIFT+F12	
Перейти к строке/столбцу...			CTRL+G	
Перейти к скобке			CTRL+SHIFT+\	
Следующая проблема			F8	
Предыдущая проблема			SHIFT+F8	
Следующее изменение			ALT+F3	
Предыдущее изменение			SHIFT+ALT+F3	

Рис. 1.17. Меню Переход

Выполнить	Терминал	Справка	Untitled-1
Запустить отладку			F5
Запуск без отладки			CTRL+F5
Остановить отладку			SHIFT+F5
Перезапустить отладку			CTRL+SHIFT+F5
Открыть конфигурации			
Добавить конфигурацию...			
Шаг с обходом			F10
Шаг с заходом			F11
Шаг с выходом			SHIFT+F11
Продолжить			F5
Переключить точку останова			F9
Новая точка останова			>
Включить все точки останова			
Отключить все точки останова			
Удалить все точки останова			
Установить дополнительные отладчики...			

Рис. 1.18. Меню Выполнить

Терминал	Справка	Untitled-1 - Visual Studio Code
Создать терминал		CTRL+SHIFT+`
Разделить терминал		CTRL+SHIFT+5
Запуск задачи...		
Запустить задачу сборки...		CTRL+SHIFT+B
Запустить активный файл		
Запустить выбранный текст		
Показать выполняющиеся задачи...		
Перезапустить выполняющуюся задачу...		
Завершить задачу...		
Настройка задач...		
Настроить задачу сборки по умолчанию...		

Рис. 1.19. Меню Терминал

- ❑ Меню **Переход** — команды для осуществления переходов в различные места редактируемого кода (рис. 1.17).
- ❑ Меню **Выполнить** — команды отладки (рис. 1.18).
- ❑ Меню **Терминал** — команды для управления терминалом (рис. 1.19).
- ❑ Меню **Справка** — доступ к справочной информации (рис. 1.20).

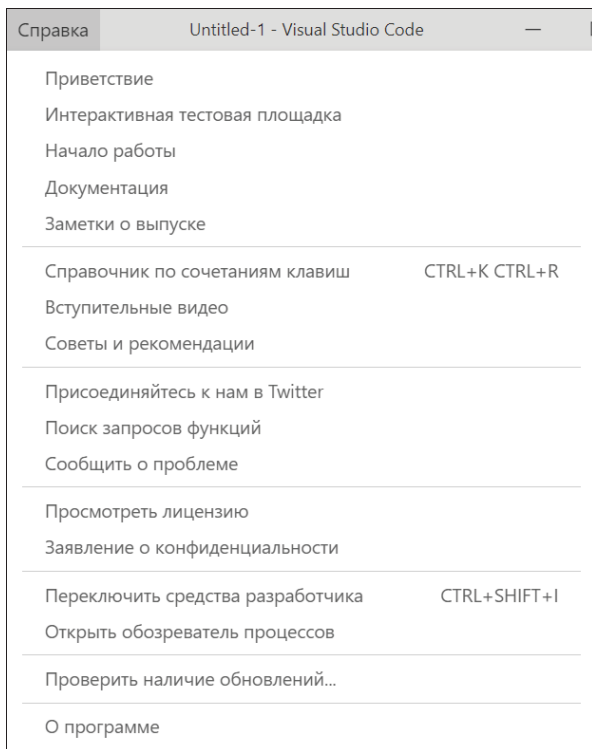


Рис. 1.20. Меню Справка

1.4.2. Боковая панель и панель действий

К левому краю окна прикреплена панель действий, имеющая вид вертикальной темной полосы с набором значков (см. рис. 1.9). При щелчке на любом значке появится боковая панель, посредством которой можно работать с файлами и каталогами, выполнять поиск в файлах, взаимодействовать с Git и др. Повторный щелчок на том же значке панели действий приведет к сокрытию боковой панели. Кратко рассмотрим значки на боковой панели.

- ❑ **Проводник** — показывает панель для работы с файлами, каталогами и проектами. Для отображения панели также можно в меню **Вид** (см. рис. 1.14) выбрать пункт **Проводник** или нажать комбинацию клавиш <Shift>+<Ctrl>+<E>.
- ❑ **Поиск** — показывает панель поиска (также выводится выбором в меню **Вид** пункта **Поиск** или нажатием комбинации клавиш <Shift>+<Ctrl>+<F>).

- ❑ **Система управления версиями** — показывает панель для работы с Git (также выводится выбором в меню **Вид** пункта **Диспетчер служб** или комбинацией клавиш <Shift>+<Ctrl>+<G>).
- ❑ **Запуск и отладка** — показывает панель отладки кода (также выводится выбором в меню **Вид** пункта **Выполнить** или комбинацией клавиш <Shift>+<Ctrl>+<D>).
- ❑ **Расширения** — показывает панель управления расширениями (также выводится выбором в меню **Вид** пункта **Расширения** или комбинацией клавиш <Shift>+<Ctrl>+<X>).
- ❑ **Учетные записи** — позволяет управлять учетными записями.
- ❑ **Управление** — отображает меню быстрого доступа к настройкам редактора (рис. 1.21).

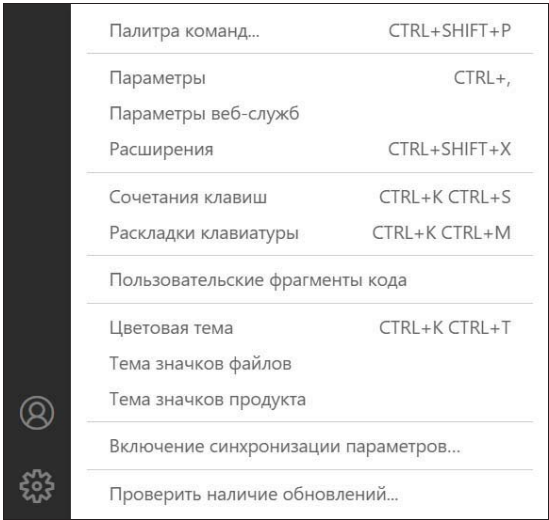


Рис. 1.21. Меню быстрого доступа к настройкам редактора

Для вывода боковой панели также можно в меню **Вид** выбрать пункт **Внешний вид | Показать боковую панель** (см. рис. 1.15) или нажать комбинацию клавиш <Ctrl>+.

Для сокрытия или отображения панели действий в меню **Вид** выбираем пункт **Внешний вид | Показать панель действий** (см. рис. 1.15). Чтобы прикрепить боковую панель к правой стороне окна, в том же меню выбираем пункт **Переместить боковую панель вправо**. Если щелкнуть правой кнопкой мыши на боковой панели, то в контекстном меню можно будет отключить отображение какого-либо значка.

1.4.3. Строка состояния

В нижней части окна (см. рис. 1.9) расположена строка состояния, в которой выводится различная служебная информация. Сейчас у нас никакой файл не открыт на

редактирование, поэтому информация о файле не отображается. Доступны лишь значки с количеством предупреждающих и критических проблем, а также значок для доступа к уведомлениям.

Для сокрытия или отображения строки состояния в меню **Вид** выбираем пункт **Внешний вид | Показать строку состояния** (см. рис. 1.15). Если щелкнуть правой кнопкой мыши на строке состояния, то в контекстном меню можно будет отключить вывод какой-либо информации или скрыть строку состояния.

1.4.4. Нижняя панель

Чтобы отобразить нижнюю панель, достаточно щелкнуть на значках с количеством предупреждающих и критических проблем в строке состояния. В итоге отобразится вкладка **Проблемы** на нижней панели. Можно также в меню **Вид** выбрать пункт **Внешний вид | Показать панель** (см. рис. 1.15) или нажать комбинацию клавиш **<Ctrl>+<J>**. Нижняя панель содержит следующие основные вкладки:

- ☐ **Проблемы** — описание предупреждающих и критических проблем. Для отображения вкладки также можно в меню **Вид** (см. рис. 1.14) выбрать пункт **Проблемы** или нажать комбинацию клавиш **<Shift>+<Ctrl>+<M>**;
- ☐ **Выходные данные** — для отображения вкладки также можно в меню **Вид** выбрать пункт **Выходные данные** или нажать **<Shift>+<Ctrl>+<U>**;
- ☐ **Терминал** — позволяет выполнять команды. Для вывода вкладки также можно в меню **Вид** выбрать пункт **Терминал** или нажать **<Ctrl>+<обратный апостроф>**;
- ☐ **Консоль отладки** — для отображения вкладки также можно в меню **Вид** выбрать пункт **Консоль отладки** или нажать **<Shift>+<Ctrl>+<Y>**.

Более подробно эти вкладки мы будем рассматривать по мере изучения материала. Способы отображения вкладки **Терминал** (рис. 1.22) нужно знать наизусть, т. к. мы будем использовать ее для выполнения команд вместо командной строки.

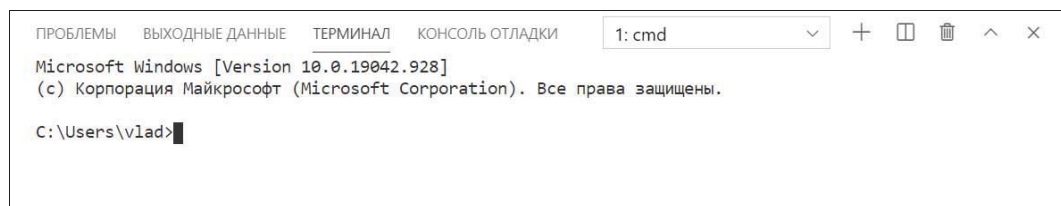


Рис. 1.22. Нижняя панель (вкладка **Терминал**)

1.4.5. Палитра команд

Если в меню **Вид** выбрать пункт **Палитра команд** или нажать комбинацию клавиш **<Shift>+<Ctrl>+<P>**, отобразится текстовое поле с выпадающим списком, который содержит все доступные в редакторе команды. Для поиска команды можно ввести

ее название в текстовое поле. Выбираем нужный пункт из списка и следуем инструкциям, которые зависят от команды.

Палитра команд открывается также при щелчке на значках в строке состояния. Например, с ее помощью можно сменить кодировку файла, указать тип файла и комбинацию символов для перевода строки. Если в меню **Файл** выбрать пункт **Настройки | Пользовательские фрагменты кода**, то с помощью палитры команд (см. рис. 1.29) можно будет создать или открыть файл с пользовательскими фрагментами кода.

1.5. Работа с файлами и каталогами

Редактор VS Code позволяет работать как с отдельными файлами, так и с целой структурой файлов и каталогов. Все изменения файлов или содержимого каталога редактор отслеживает и автоматически выполняет обновление.

1.5.1. Создание и сохранение файла

Для создания нового файла в меню **Файл** выбираем пункт **Новый файл** или нажимаем комбинацию клавиш `<Ctrl>+<N>`. В результате в рабочей области будет создана вкладка с названием **Untitled-1**.

Чтобы сохранить файл, в меню **Файл** выбираем пункт **Сохранить как** или нажимаем комбинацию клавиш `<Shift>+<Ctrl>+<S>`. В открывшемся окне выбираем каталог (например, `C:\book`), в поле **Имя файла** вводим имя файла, а в списке **Тип файла** указываем тип сохраняемого файла (например, **HTML**).

Сохраним созданный ранее "пустой" файл под именем `index.html` в каталоге `C:\book`. В результате в заголовке вкладки отобразится имя файла с расширением. При наведении указателя мыши на заголовок вкладки отобразится всплывающая подсказка с полным путем к файлу.

Обратите внимание на строку состояния. Там отобразился формат файла: **HTML**; символы переноса строк: **LF** (соответствует комбинации символов `\n`); кодировка файла: **UTF-8** (без служебных символов BOM), и другая служебная информация. Если щелкнуть на этих надписях, то можно изменить формат файла, кодировку и символы переноса строк.

Запишем в файл структуру HTML-документа. Для этого достаточно ввести восклицательный знак и нажать клавишу `<Tab>`. В результате будет вставлен шаблон, приведенный в листинге 1.1.

Листинг 1.1. Шаблон структуры HTML-документа

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
```



```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Document</title>
</head>
<body>

</body>
</html>
```

Мы нажали всего две клавиши, а получили целую структуру HTML-документа — и все это благодаря расширению Emmet, входящему в состав VS Code. Возможности этого расширения мы рассмотрим немного позже в этой главе.

Чтобы сохранить исправленный файл, в меню **Файл** выбираем пункт **Сохранить** или нажимаем комбинацию клавиш **<Ctrl>+<S>**. Если вы выполнили инструкции по настройке редактора из *разд. 1.2*, то при потере вкладкой фокуса ввода файл будет сохранен автоматически. В этом случае вы не сможете забыть сохранить файл перед его запуском и избежите множества проблем.

Сохранить все исправленные файлы можно, выбрав в меню **Файл** пункт **Сохранить все** или нажав комбинацию клавиш **<Ctrl>+<K>**, затем **<S>**.

Если какие-либо файлы еще не сохранены, то на значке **Проводник** на панели действий будет показан синий круг с числом белого цвета, означающим количество несохраненных файлов. После сохранения файла круг исчезает. Так что обращайтесь на панель действий, особенно если что-то не получается.

Если раскрыть боковую панель **Проводник**, то в разделе **Открытые редакторы** можно увидеть открытый файл. Когда открыто несколько файлов, с помощью этого раздела можно быстро переключиться на нужный файл. В разделе **Структура** отображается структура HTML-документа в виде дерева. При щелчке на элементе этой структуры текстовый курсор перескочит на строку исходного кода, соответствующую выбранному элементу.

Если в боковой панели **Проводник** отсутствуют разделы **Открытые редакторы** и (или) **Структура**, щелкните на кнопку со значком многоточия, которая находится правее надписи "Проводник" сверху панели. Появится небольшое меню, посредством которого можно вывести отсутствующий раздел на экран.

1.5.2. Заккрытие вкладки с файлом

Заккрыть вкладку с файлом можно следующими основными способами:

- ☐ нажать на значок в виде крестика в заголовке вкладки;
- ☐ на боковой панели **Проводник** нажать на значок в виде крестика слева от строки с именем файла;
- ☐ в меню **Файл** выбрать пункт **Заккрыть редактор**. При этом закрываемая вкладка должна быть текущей;
- ☐ нажать комбинацию клавиш **<Ctrl>+<F4>**. При этом закрываемая вкладка должна быть текущей.

1.5.3. Открытие файла

Открыть файл можно следующими основными способами:

- ☐ в меню **Файл** выбрать пункт **Открыть файл**, в открывшемся окне указать нужный файл и нажать кнопку **Открыть**;
- ☐ нажать комбинацию клавиш `<Ctrl>+<O>`, в открывшемся окне выбрать нужный файл и нажать кнопку **Открыть**;
- ☐ в Проводнике Windows щелкнуть правой кнопкой мыши на нужном файле и из контекстного меню выбрать пункт **Открыть с помощью Code**. Пункт будет доступен, если при установке вы отметили соответствующий флажок (см. рис. 1.4);
- ☐ перетащить файл из Проводника Windows в окно редактора и, как только цвет фона окна изменится на синий, сбросить файл. Это самый простой и наиболее часто используемый способ открытия файла;
- ☐ в командной строке Windows ввести команду `code` и через пробел имя открываемого файла:

```
C:\book>code index.html
```

1.5.4. Открытие каталога

Наиболее часто мы будем работать с множеством файлов, хранящихся в одном каталоге (и, скорее всего, распределенных по вложенным каталогам), т. е. с проектом. Открыть каталог проекта в редакторе VS Code можно следующими основными способами:

- ☐ в меню **Файл** выбрать пункт **Открыть папку**, в открывшемся окне выбрать нужный каталог и нажать кнопку **Выбор папки**;
- ☐ нажать комбинацию клавиш `<Ctrl>+<K>`, затем `<Ctrl>+<O>`, в открывшемся окне выбрать нужный каталог и нажать кнопку **Выбор папки**;
- ☐ на боковой панели **Проводник** нажать кнопку **Открыть папку**, в открывшемся окне выбрать каталог и нажать кнопку **Выбор папки**;
- ☐ в Проводнике Windows щелкнуть правой кнопкой мыши на каталоге или на свободном месте внутри каталога и из контекстного меню выбрать пункт **Открыть с помощью Code**. Пункт будет доступен, если при установке вы отметили соответствующий флажок (см. рис. 1.4);
- ☐ перетащить каталог из Проводника Windows в окно редактора и, как только цвет фона окна изменится на синий, сбросить каталог. Это самый простой и наиболее часто используемый способ открытия каталога;
- ☐ если каталог открывался ранее, то в меню **Файл** выбрать пункт **Открыть последние** и выбрать каталог в появившемся меню.

На боковой панели **Проводник** отобразится имя и содержимое открытого каталога. Если щелкнуть на имени файла, он будет открыт в отдельной вкладке, а в разделе

Открытые редакторы появятся его имя (рис. 1.23). После закрытия вкладки имя файла будет удалено из раздела **Открытые редакторы**, но файл все равно будет доступен в списке файлов открытого каталога. Таким образом, Проводник Windows нам больше не нужен — все действия по созданию файлов и каталогов, просмотру их содержимого можно выполнять внутри редактора.

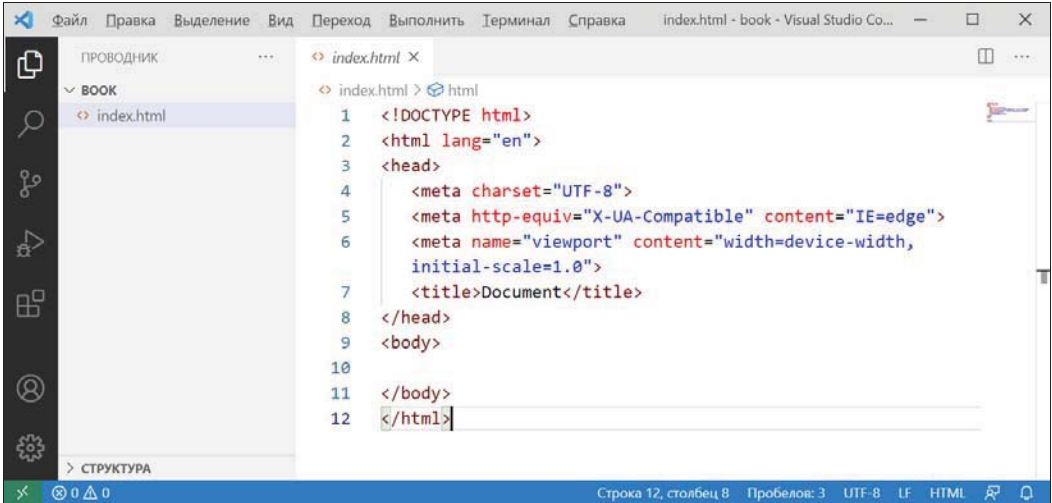


Рис. 1.23. Редактор с открытым каталогом и содержимым файла на отдельной вкладке

На боковой панели **Проводник** в заголовке раздела с каталогом расположены четыре кнопки (см. рис. 1.23):

- ❑ **Создать файл** — создает новый файл в текущем каталоге. Чтобы сделать каталог текущим, достаточно щелкнуть на нем.

После нажатия кнопки в каталоге появится текстовое поле, в которое следует ввести имя нового файла с расширением (например, `app.js`) и нажать клавишу `<Enter>`. Содержимое созданного файла будет открыто в новой вкладке для редактирования.

В контекстном меню панели доступен пункт **Создать файл** с аналогичным назначением;

- ❑ **Новая папка** — создает новый каталог в текущем каталоге.

После нажатия этой кнопки в каталоге появится текстовое поле, в которое следует ввести имя нового каталога (например, `css`) и нажать клавишу `<Enter>`.

В контекстном меню панели доступен пункт **Новая папка** с аналогичным назначением;

- ❑ **Обновить окно проводника** — эту кнопку следует нажать, если были выполнены какие-либо действия с файлами вне редактора VS Code и редактор не "видит" изменений;
- ❑ **Свернуть папки в проводнике** — скрывает содержимое вложенных каталогов.

Чтобы переименовать файл или каталог, нужно щелкнуть на нем правой кнопкой мыши и из контекстного меню выбрать пункт **Переименование**. Также можно выбрать файл (каталог) и нажать клавишу <F2>. Отобразится текстовое поле, в котором нужно внести изменения и нажать клавишу <Enter>. Файл можно вырезать или скопировать, а затем вставить с помощью соответствующих пунктов из контекстного меню или клавиш быстрого доступа.

Чтобы удалить файл или каталог, нужно щелкнуть на нем правой кнопкой мыши и из контекстного меню выбрать пункт **Удалить**. Также можно выбрать файл (каталог) и нажать клавишу <Delete>. В результате отобразится диалоговое окно, в котором нужно подтвердить удаление (рис. 1.24).

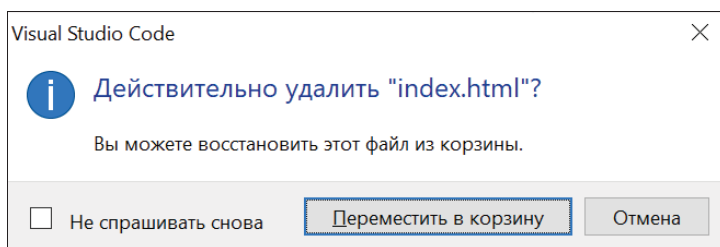


Рис. 1.24. Диалоговое окно для подтверждения удаления файла

Заккрыть каталог можно следующими основными способами:

- ☐ в меню **Файл** выбрать пункт **Заккрыть папку**;
- ☐ нажать комбинацию клавиш <Ctrl>+<K>, а затем <F>.

Все действия, рассмотренные в этом разделе, нужно знать наизусть. В дальнейшем мы будем говорить "откройте каталог в редакторе", "создайте файл" или "создайте каталог", "откройте файл на редактирование" без явного указания, как это сделать.

1.6. Отображение содержимого нескольких файлов одновременно

В предыдущем разделе мы создали три файла: index.html, app.js и main.css во вложенном каталоге css. Давайте подключим два последних файла к HTML-документу (листинг 1.2).

Листинг 1.2. Содержимое файла index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet" type="text/css" href="css/main.css">
```

```

<title>Заголовок страницы</title>
</head>
<body>
  <div id="div1">Привет, мир!</div>
  <script src="app.js"></script>
</body>
</html>

```

Содержимое каждого файла открывается на отдельной вкладке, а имя файла отображается в заголовке вкладки. В каждый момент времени показывается содержимое только одной вкладки. Чтобы переключиться на другую вкладку, достаточно щелкнуть мышью на ее заголовке.

При создании веб-страницы мы часто работаем с несколькими файлами, и было бы удобно видеть содержимое сразу нескольких файлов (например, страницы и таблицы стилей). Редактор VS Code позволяет нам это сделать (рис. 1.25). Разделить окно редактора на два поля можно следующими способами:

- ❑ активизировать вкладку и в строке заголовков вкладок щелкнуть на значке **Разбить содержимое окна редактора вправо** или нажать комбинацию клавиш `<Ctrl>+<I>`. Содержимое окна будет разделено на два поля по горизонтали, активная вкладка появится в правом поле;
- ❑ щелкнуть правой кнопкой мыши на заголовке вкладки и из контекстного меню выбрать один из пунктов:
 - **Разбить содержимое окна редактора вправо** — содержимое окна будет разделено на два поля по горизонтали, активная вкладка появится в правом поле;

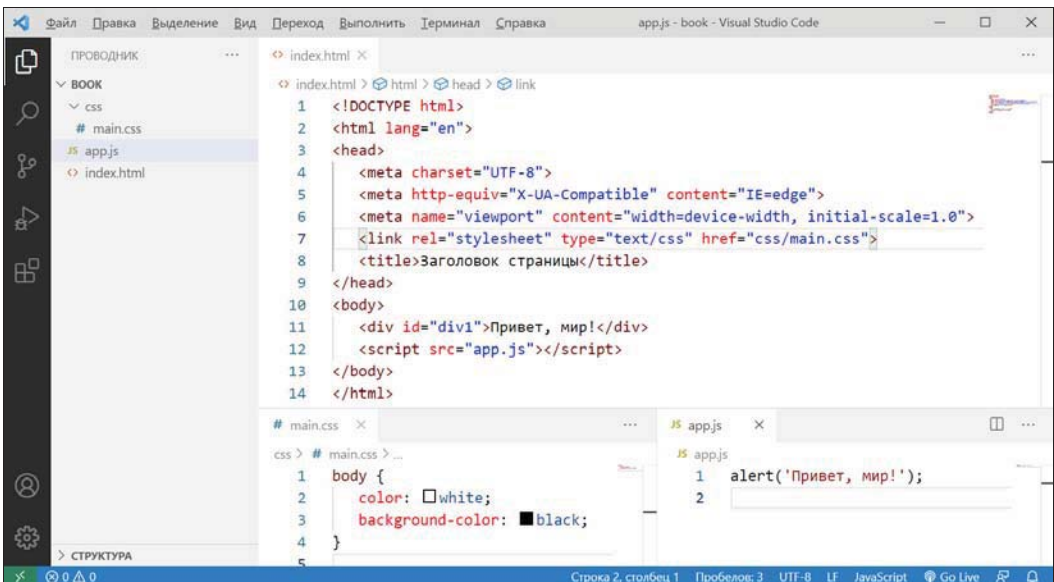


Рис. 1.25. Отображение содержимого нескольких файлов одновременно

- **Разбить содержимое окна редактора влево** — то же самое, только активная вкладка появится в левом поле;
- **Разбить содержимое окна редактора вверх** — содержимое окна будет разделено на два поля по вертикали, активная вкладка появится в верхнем поле;
- **Разбить содержимое окна редактора вниз** — то же самое, только активная вкладка появится в нижнем поле.

Эти же пункты доступны во вложенном меню, появляющемся после выбора в меню **Вид** пункта **Макет редактора** (см. рис. 1.16);

- ☐ перетащить вкладку мышью в нужное место. Поле, в которое будет помещена перемещаемая вкладка, подсвечивается.

Можно перетаскивать мышью заголовки вкладок из одной рабочей области в другую или менять порядок их следования. Если закрыть последнюю вкладку, поле будет удалено.

1.7. Live Server: автоматическое обновление веб-страницы

После изменения любого из трех файлов нужно перейти в веб-браузер и вручную обновить страницу. В этой книге мы будем пользоваться веб-браузером Mozilla Firefox.

Обновить веб-страницу в Firefox можно следующими способами:

- ☐ нажатием кнопки **Обновить текущую страницу** в адресной строке;
- ☐ щелчком правой кнопкой мыши на заголовке текущей вкладки и выбором из контекстного меню пункта **Обновить вкладку**;
- ☐ нажатием клавиши <F5> или комбинации клавиш <Ctrl>+<R>.

Существует возможность автоматического обновления веб-страницы сразу после сохранения файла. Для этого нужно установить расширение Live Server. Переходим в панель **Расширения**, нажав комбинацию клавиш <Shift>+<Ctrl>+<X>, и в поле для поиска вводим "Live Server". Находим расширение в списке и устанавливаем его. После установки в строке состояния появится значок с надписью **Go Live** (см. рис. 1.25).

Делаем активной вкладку с файлом index.html и в строке состояния щелкаем на надписи **Go Live**. Можно также щелкнуть правой кнопкой мыши внутри содержимого вкладки и из контекстного меню выбрать пункт **Open with Live Server**. В результате запустится сервер и в текущем веб-браузере откроется веб-страница, расположенная по интернет-адресу **http://127.0.0.1:5500/index.html**. В строке состояния отобразится надпись **Port: 5500**. Теперь при любом изменении файлов index.html, app.js или main.css открытая веб-страница будет обновлена автоматически.

Для остановки сервера в строке состояния щелкаем на надписи **Port: 5500**. Можно также щелкнуть правой кнопкой мыши внутри содержимого вкладки с файлом

index.html и из контекстного меню выбрать пункт **Stop Live Server**. В строке состояния отобразится надпись **Go Live**.

1.8. Emmet: ускорение набора HTML-кода

Расширение Emmet позволяет значительно ускорить набор кода путем ввода специальной комбинации символов и нажатия клавиши `<Tab>`. Вместо этой комбинации будет вставлен фрагмент кода. Расширение не нуждается в отдельной установке, т. к. входит в состав редактора VS Code.

ПРИМЕЧАНИЕ

Комбинации, заканчивающиеся на `+` (например, `ul+`), в VS Code не работают.

1.8.1. Вставка структуры HTML-документа

Для вставки структуры HTML-документа достаточно ввести восклицательный знак и нажать клавишу `<Tab>`. В результате будет вставлен код, приведенный в листинге 1.1. Внутри шаблона помечены точки останова, перемещаться между которыми можно с помощью клавиши `<Tab>`:

```
!<Tab>           => <Структура HTML-документа>
html:5<Tab>       => <Структура HTML-документа>
doc<Tab>          => <Структура HTML-документа без тега doctype>
html:xml<Tab>     => <html xmlns="http://www.w3.org/1999/xhtml"></html>
```

Три восклицательных знака позволяют вставить тег `<!doctype>`:

```
!!!<Tab>         => <!DOCTYPE html>
```

В структуре HTML-документа в параметре `lang` тега `<html>` по умолчанию указывается код английского языка. Чтобы выводился код русского языка, нужно в настройках редактора выполнить поиск настройки **Emmet: Variables** и указать значение `ru` для переменной `lang` или просто добавить следующий фрагмент кода в конец файла `settings.json` (см. *павд. 1.2*):

```
"emmet.variables": {
  "lang": "ru"
}
```

Результат после изменения значений переменных при вводе комбинации `!<Tab>` показан в листинге 1.3.

Листинг 1.3. Шаблон структуры HTML-документа с указанным кодом русского языка

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
```

```

    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>

</body>
</html>

```

1.8.2. Вставка HTML-комментария

Буква **c** позволяет вставить HTML-комментарий:

```

c<Tab>          => <!-- -->
cc:ie<Tab>       => <!--[if IE]><![endif]-->
cc:ie6<Tab>      => <!--[if lte IE 6]><![endif]-->
cc:noie<Tab>     => <!--[if !IE]><!--><!--<![endif]-->

```

Чтобы закомментировать блок кода, нужно предварительно его выделить, а затем в меню **Правка** выбрать пункт **Переключить комментарий строки** или нажать комбинацию клавиш **<Ctrl>+</>**. Повторное действие удаляет комментарий. Можно также выбрать пункт **Переключить комментарий блока** или нажать комбинацию клавиш **<Shift>+<Alt>+<A>**.

1.8.3. Теги из секции заголовка (<head>)

Комбинации для тега **<meta>**:

```

meta<Tab>          => <meta>
meta:desc<Tab>     => <meta name="description" content="">
meta:kw<Tab>        => <meta name="keywords" content="">
meta:compat<Tab>    => <meta http-equiv="X-UA-Compatible" content="IE=7">
meta:edge<Tab>      => <meta http-equiv="X-UA-Compatible" content="ie=edge">
meta:redirect<Tab>  =>
    <meta http-equiv="refresh" content="0; url=http://example.com">
meta:utf<Tab>       =>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
meta:vp<Tab>        =>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">

```

Комбинации для тега **<link>**:

```

link<Tab>          => <link rel="stylesheet" href="">
link:css<Tab>       => <link rel="stylesheet" href="style.css">
link:favicon<Tab>   =>
    <link rel="shortcut icon" type="image/x-icon" href="favicon.ico">
link:im<Tab>        => <link rel="import" href="component.html">
link:import<Tab>    => <link rel="import" href="component.html">
link:manifest<Tab>  => <link rel="manifest" href="manifest.json">
link:mf<Tab>        => <link rel="manifest" href="manifest.json">
link:print<Tab>     => <link rel="stylesheet" href="print.css" media="print">

```



```

link:rss<Tab>    =>
    <link rel="alternate" type="application/rss+xml" title="RSS"
        href="rss.xml">
link:touch<Tab> => <link rel="apple-touch-icon" href="favicon.png">
link:atom<Tab>   =>
    <link rel="alternate" type="application/atom+xml" title="Atom"
        href="atom.xml">

```

Комбинации для тега `<script>`:

```

script<Tab>      => <script></script>
script:src<Tab>  => <script src=""></script>

```

Комбинация для тега `<base>`:

```

base<Tab>        => <base href="">

```

1.8.4. Основные теги из секции тела (`<body>`)

При вводе имен тегов, не имеющих отдельных комбинаций, получим открывающий и закрывающий теги:

```

div<Tab>         => <div></div>
span<Tab>        => <span></span>
p<Tab>           => <p></p>

```

Для некоторых тегов генерируются параметры, а также учитывается, является ли тег парным или одинарным:

```

abbr<Tab>        => <abbr title=""></abbr>
area<Tab>        => <area shape="" coords="" href="" alt="">
area:c<Tab>      => <area shape="circle" coords="" href="" alt="">
area:d<Tab>      => <area shape="default" coords="" href="" alt="">
area:p<Tab>      => <area shape="poly" coords="" href="" alt="">
area:r<Tab>      => <area shape="rect" coords="" href="" alt="">
audio<Tab>       => <audio src=""></audio>
br<Tab>          => <br>
map<Tab>         => <map name=""></map>
marquee<Tab>     => <marquee behavior="" direction=""></marquee>
video<Tab>       => <video src=""></video>

```

Комбинации для тега `<a>`:

```

a<Tab>           => <a href=""></a>
a:link<Tab>      => <a href="http://"></a>
a:mail<Tab>      => <a href="mailto:"></a>
a:tel<Tab>       => <a href="tel:+"></a>
a:blank<Tab>     =>
    <a href="http://" target="_blank" rel="noopener noreferrer"></a>

```

Комбинации для вставки изображений:

```

img<Tab>         => <img src="" alt="">
img:s<Tab>       => <img srcset="" src="" alt="">

```

```

img:srcset<Tab> => <img srcset="" src="" alt="">
ri:d<Tab>      => <img srcset="" src="" alt="">
ri:dpr<Tab>    => <img srcset="" src="" alt="">
img:sizes<Tab> => <img sizes="" srcset="" src="" alt="">
img:z<Tab>     => <img sizes="" srcset="" src="" alt="">
ri:v<Tab>      => <img sizes="" srcset="" src="" alt="">
ri:viewport<Tab> => <img sizes="" srcset="" src="" alt="">
pic<Tab>       => <picture></picture>
src<Tab>       => <source>
source<Tab>    => <source>
src:m<Tab>      => <source media="(min-width: )" srcset="">
source:media<Tab> => <source media="(min-width: )" srcset="">
src:mz<Tab>     =>
    <source media="(min-width: )" sizes="" srcset="">
source:media:sizes<Tab> =>
    <source media="(min-width: )" sizes="" srcset="">
src:mt<Tab>     =>
    <source media="(min-width: )" srcset="" type="image/">
source:media:type<Tab> =>
    <source media="(min-width: )" srcset="" type="image/">
src:z<Tab>      => <source sizes="" srcset="">
source:sizes<Tab> => <source sizes="" srcset="">
src:zt<Tab>     => <source sizes="" srcset="" type="image/">
source:sizes:type<Tab> => <source sizes="" srcset="" type="image/">
src:sc<Tab>     => <source src="" type="">
source:src<Tab> => <source src="" type="">
src:s<Tab>      => <source srcset="">
source:srcset<Tab> => <source srcset="">
src:t<Tab>      => <source srcset="" type="image/">
source:type<Tab> => <source srcset="" type="image/">
ri:a<Tab> или ri:art<Tab> =>
    <picture>
        <source media="(min-width: )" srcset="">
        <img src="" alt="">
    </picture>
ri:t<Tab> или ri:type<Tab> =>
    <picture>
        <source srcset="" type="image/">
        <img src="" alt="">
    </picture>

```

В следующих случаях можно использовать сокращенные имена тегов:

```

mn<Tab>      => <main></main>
hdr<Tab>     => <header></header>
sect<Tab>    => <section></section>
art<Tab>     => <article></article>
ftr<Tab>     => <footer></footer>
adr<Tab>     => <address></address>

```

```

bq<Tab>      => <blockquote></blockquote>
emb<Tab>      => <embed src="" type="">
fig<Tab>      => <figure></figure>
figc<Tab>     => <figcaption></figcaption>
ifr<Tab>      => <iframe src="" frameborder="0"></iframe>
str<Tab>      => <strong></strong>
det<Tab>      => <details></details>
det>summary+p<Tab> =>
    <details>
        <summary></summary>
        <p></p>
    </details>

```

1.8.5. Добавление текста

После вставки шаблона тега без параметров текстовый курсор будет расположен перед закрывающим тегом, что позволяет сразу начать набор текста:

```
p<Tab>      => <p>{Курсор}</p>
```

Текст можно также добавить в составе комбинации внутри фигурных скобок:

```

p{Текст}<Tab> => <p>Текст</p>
p{Текст1 }>a{ссылка}+{ текст2}<Tab> =>
    <p>Текст1 <a href="">ссылка</a> текст2</p>

```

Если в составе текста указать символ \$, то вместо него будет подставлен порядковый номер элемента. Создадим три абзаца и пронумеруем их:

```

p*3>{Абзац $}<Tab> =>
    <p>Абзац 1</p>
    <p>Абзац 2</p>
    <p>Абзац 3</p>

```

Если указать несколько символов \$ подряд, то перед номером добавятся нули:

```

p*3>{Абзац $$$}<Tab> =>
    <p>Абзац 001</p>
    <p>Абзац 002</p>
    <p>Абзац 003</p>

```

По умолчанию нумерация элементов начинается с 1. После символа @ можно указать другой начальный номер, например 3:

```

p*3>{Абзац $@3}<Tab> =>
    <p>Абзац 3</p>
    <p>Абзац 4</p>
    <p>Абзац 5</p>

```

Для вставки произвольного текста-заполнителя следует ввести слово "lorem", после которого можно указать необязательное число генерируемых слов. Создадим три абзаца и добавим в них по пять слов:

```
p*3>lorem5<Tab> =>
  <p>Lorem ipsum dolor sit amet.</p>
  <p>Quidem dolore odio explicabo iste!</p>
  <p>Inventore repellat doloribus blanditiis fugit!</p>
```

После слова "lorem" можно указать код языка. Сгенерируем текст-заполнитель на русском языке:

```
p>loremru20<Tab> =>
  <p>Далеко-далеко за словесными горами в стране гласных и согласных
  живут рыбные тексты. Сих его, предупреждал залетают вдали рыбными
  заглавных встретил текст щеке.</p>
```

1.8.6. Добавление параметров тегов

Если указать только CSS-селектор стилевого класса или идентификатора, то будет создан тег <div> с переданным стилевым классом или идентификатором:

```
.row<Tab>          => <div class="row"></div>
.row.mb-3<Tab>     => <div class="row mb-3"></div>
#div1<Tab>         => <div id="div1"></div>
#div1.row<Tab>     => <div id="div1" class="row"></div>
```

Чтобы добавить эти параметры в другой тег, предварительно указываем его имя:

```
p.text-center<Tab> => <p class="text-center"></p>
```

Если эти параметры вкладываются в блочный тег, то они добавляются к тегу :

```
p>.text-center<Tab> => <p><span class="text-center"></span></p>
```

или к другим вложенным тегам в зависимости от родительского элемента:

```
ul>.item<Tab>      =>
  <ul>
    <li class="item"></li>
  </ul>
table>.row>.col<Tab> =>
  <table>
    <tr class="row">
      <td class="col"></td>
    </tr>
  </table>
```

Произвольные параметры следует указывать внутри квадратных скобок. Значение параметра задается после символа = внутри одинарных или двойных кавычек. Если значение не содержит пробела, то кавычки можно не указывать. Если параметр не содержит значения, то достаточно вставить только имя параметра. Пример:

```
p[title="Подсказка" class=p$]*2<Tab> =>
  <p title="Подсказка" class="p1"></p>
  <p title="Подсказка" class="p2"></p>
p[title]<Tab> => <p title=""></p>
```

1.8.7. Формы и элементы управления

Для вставки форм и элементов управления предусмотрены следующие комбинации символов:

```

form<Tab>          => <form action=""></form>
form:get<Tab>      => <form action="" method="get"></form>
form:post<Tab>     => <form action="" method="post"></form>

label<Tab>         => <label for=""></label>

input:s<Tab>       => <input type="submit" value="">
input:submit<Tab>  => <input type="submit" value="">
input:reset<Tab>   => <input type="reset" value="">
input:b<Tab>       => <input type="button" value="">
input:button<Tab>  => <input type="button" value="">
input:i<Tab>       => <input type="image" src="" alt="">
input:image<Tab>   => <input type="image" src="" alt="">
btn<Tab>           => <button></button>
button<Tab>        => <button></button>
btn:s<Tab>         => <button type="submit"></button>
button:s<Tab>      => <button type="submit"></button>
button:submit<Tab> => <button type="submit"></button>
btn:r<Tab>         => <button type="reset"></button>
button:r<Tab>      => <button type="reset"></button>
button:reset<Tab>  => <button type="reset"></button>
btn:d<Tab>         => <button disabled="disabled"></button>
button:d<Tab>      => <button disabled="disabled"></button>
button:disabled<Tab> => <button disabled="disabled"></button>

tarea<Tab>        => <textarea name="" id="" cols="30" rows="10"></textarea>
textarea<Tab>     => <textarea name="" id="" cols="30" rows="10"></textarea>

input<Tab>         => <input type="text">
inp<Tab>           => <input type="text" name="" id="">
input:t<Tab>       => <input type="text" name="" id="">
input:text<Tab>    => <input type="text" name="" id="">
input:p<Tab>       => <input type="password" name="" id="">
input:password<Tab> => <input type="password" name="" id="">
input:url<Tab>     => <input type="url" name="" id="">
input:email<Tab>   => <input type="email" name="" id="">
input:tel<Tab>     => <input type="tel" name="" id="">
input:number<Tab>  => <input type="number" name="" id="">
input:search<Tab>  => <input type="search" name="" id="">
input:date<Tab>    => <input type="date" name="" id="">
input:time<Tab>    => <input type="time" name="" id="">
input:datetime<Tab> => <input type="datetime" name="" id="">

```

```

input:datetime-local<Tab> => <input type="datetime-local" name="" id="">
input:month<Tab>      => <input type="month" name="" id="">
input:week<Tab>       => <input type="week" name="" id="">
input:h<Tab>          => <input type="hidden" name="">
input:hidden<Tab>     => <input type="hidden" name="">
datal<Tab>            => <datalist></datalist>

select<Tab>           => <select name="" id=""></select>
select:d<Tab>          => <select name="" id="" disabled="disabled"></select>
select:disabled<Tab> =>
    <select name="" id="" disabled="disabled"></select>
opt<Tab>               => <option value=""></option>
option<Tab>            => <option value=""></option>
optg<Tab>              => <optgroup></optgroup>
optgroup<Tab>          => <optgroup></optgroup>
select>option*3<Tab> =>
    <select name="" id="">
        <option value=""></option>
        <option value=""></option>
        <option value=""></option>
    </select>

input:c<Tab>           => <input type="checkbox" name="" id="">
input:checkbox<Tab>       => <input type="checkbox" name="" id="">
input:r<Tab>           => <input type="radio" name="" id="">
input:radio<Tab>       => <input type="radio" name="" id="">

input:f<Tab>           => <input type="file" name="" id="">
input:file<Tab>        => <input type="file" name="" id="">
input:range<Tab>       => <input type="range" name="" id="">
input:color<Tab>       => <input type="color" name="" id="">
prog<Tab>              => <progress></progress>

fst<Tab>               => <fieldset></fieldset>
fset<Tab>              => <fieldset></fieldset>
fst:d<Tab>             => <fieldset disabled="disabled"></fieldset>
fset:d<Tab>            => <fieldset disabled="disabled"></fieldset>
leg<Tab>               => <legend></legend>
legend<Tab>           => <legend></legend>

```

1.8.8. Таблицы

Для быстрой вставки таблиц, строк и ячеек можно использовать такие комбинации:

```

table<Tab>             => <table></table>
cap<Tab>               => <caption></caption>
tr<Tab>                => <tr></tr>

```

```

th<Tab>          => <th></th>
td<Tab>          => <td></td>
thead<Tab>       => <thead></thead>
tbody<Tab>       => <tbody></tbody>
tfoot<Tab>       => <tfoot></tfoot>
colg<Tab>        => <colgroup></colgroup>
col<Tab>         => <col>

```

```
table>tr*2>td*2<Tab> =>
```

```

<table>
  <tr>
    <td></td>
    <td></td>
  </tr>
  <tr>
    <td></td>
    <td></td>
  </tr>
</table>

```

```
thead>tr>th*2<Tab> =>
```

```

<thead>
  <tr>
    <th></th>
    <th></th>
  </tr>
</thead>

```

1.8.9. Списки

Для быстрой вставки списков можно применить следующие комбинации:

```

ul<Tab>          => <ul></ul>
ol<Tab>          => <ol></ol>
li<Tab>          => <li></li>

```

```
ul>li*3<Tab> =>
```

```

<ul>
  <li></li>
  <li></li>
  <li></li>
</ul>

```

```
ul>li*3>{Пункт $}<Tab> =>
```

```

<ul>
  <li>Пункт 1</li>
  <li>Пункт 2</li>
  <li>Пункт 3</li>
</ul>

```

```

ol>.item$*3<Tab> =>
  <ol>
    <li class="item1"></li>
    <li class="item2"></li>
    <li class="item3"></li>
  </ol>

dl<Tab>           => <dl></dl>
dt<Tab>           => <dt></dt>
dd<Tab>           => <dd></dd>
dl>(dt+dd)*2<Tab> =>
  <dl>
    <dt></dt>
    <dd></dd>
    <dt></dt>
    <dd></dd>
  </dl>

```

1.8.10. Вставка сразу нескольких тегов

Чтобы вставить несколько одинаковых тегов друг за другом, следует добавить символ *, после которого указать число повторов. Вставим три абзаца и пронумеруем их:

```

p*3>{Абзац $}<Tab> =>
  <p>Абзац 1</p>
  <p>Абзац 2</p>
  <p>Абзац 3</p>

```

Символ > означает, что нужно выполнить вложение внутрь тега. В предыдущем примере мы вкладывали текст. Вложим пункт внутрь списка:

```

ul>li<Tab> =>
  <ul>
    <li></li>
  </ul>

```

Символ + добавляет элемент после текущего элемента:

```

div+p+ul>li*3<Tab> =>
  <div></div>
  <p></p>
  <ul>
    <li></li>
    <li></li>
    <li></li>
  </ul>

```

Как видно из примера, можно комбинировать символы *, > и +, создавая сложные структуры. В последнем подвыражении мы вставляли пункты внутрь списка. Если

в конец выражения вставить символ + и тег, то этот тег будет вставлен после тега `<li>`. Как же нам вставить тег после списка? Чтобы переместиться выше по дереву на один уровень, нужно указать символ `^`. Пример:

```
ul>li*3^div<Tab> =>
  <ul>
    <li></li>
    <li></li>
    <li></li>
  </ul>
  <div></div>
```

Чтобы переместиться на несколько уровней выше, символ `^` следует указать несколько раз:

```
div>p>span+i^^div<Tab> =>
  <div>
    <p><span></span><i></i></p>
  </div>
  <div></div>
```

Можно также ограничить вложение, добавив группу внутри круглых скобок:

```
(div>p>span+i)<div><Tab> =>
  <div>
    <p><span></span><i></i></p>
  </div>
  <div></div>
```

1.9. Emmet: ускорение набора CSS-кода

Специальные комбинации символов, рассмотренные в предыдущих разделах, работают только в файлах с расширениями `html`, `php`, `jsx` и некоторых других. В CSS-файлах (и в файлах CSS-препроцессоров) нужно использовать другие комбинации, которые мы рассмотрим в этом разделе.

1.9.1. Форматирование шрифта

Одновременное указание характеристик шрифта:

```
f<Tab>      => font: 1em sans-serif;
```

Имя шрифта:

```
ff<Tab>     => font-family: serif;
ff:s<Tab>   => font-family: serif;
ff:ss<Tab>  => font-family: sans-serif;
ff:c<Tab>   => font-family: cursive;
ff:f<Tab>   => font-family: fantasy;
ff:m<Tab>   => font-family: monospace;
```

```
ff:a<Tab>  => font-family: auto;
ff:u<Tab>  => font-family: unset;
ff:i<Tab>  => font-family: inherit;
fft<Tab>   =>
    font-family: "Times New Roman", Times, Baskerville, Georgia, serif;
ffa<Tab>   =>
    font-family: Arial, "Helvetica Neue", Helvetica, sans-serif;
ffv<Tab>   => font-family: Verdana, Geneva, sans-serif;
```

Стиль шрифта:

```
fs<Tab>    => font-style: italic;
fs:i<Tab>  => font-style: italic;
fs:n<Tab>  => font-style: normal;
fs:o<Tab>  => font-style: oblique;
```

Размер шрифта:

```
fsz<Tab>   => font-size: ;
fsza<Tab>  => font-size-adjust: ;
```

Цвет текста:

```
c<Tab>     => color: #000;
```

Жирность шрифта:

```
fw<Tab>    => font-weight: normal;
fw:n<Tab>  => font-weight: normal;
fw:b<Tab>  => font-weight: bold;
fw:lr<Tab> => font-weight: lighter;
fw:br<Tab> => font-weight: bolder;
```

Вид строчных букв:

```
fv<Tab>    => font-variant: normal;
fv:n<Tab>  => font-variant: normal;
fv:sc<Tab> => font-variant: small-caps;
```

Загружаемые шрифты:

```
@f<Tab>    =>
    @font-face {
        font-family: ;
        src: url();
    }
@ff<Tab>   =>
    @font-face {
        font-family: 'FontName';
        src: url('FileName.eot');
        src: url('FileName.eot?#iefix') format('embedded-opentype'),
            url('FileName.woff') format('woff'),
            url('FileName.ttf') format('truetype'),
            url('FileName.svg#FontName') format('svg');
```

```

    font-style: normal;
    font-weight: normal;
}

```

Прочие атрибуты:

```

fd<Tab>      => font-display: auto;
fd:a<Tab>    => font-display: auto;
fd:b<Tab>    => font-display: block;
fd:s<Tab>    => font-display: swap;
fd:f<Tab>    => font-display: fallback;
fd:o<Tab>    => font-display: optional;
fef<Tab>     => font-effect: none;
fef:n<Tab>   => font-effect: none;
fef:eg<Tab>  => font-effect: engrave;
fef:eb<Tab>  => font-effect: emboss;
fef:o<Tab>   => font-effect: outline;
fem<Tab>     => font-emphasize: ;
femp<Tab>    => font-emphasize-position: before;
femp:b<Tab>  => font-emphasize-position: before;
femp:a<Tab>  => font-emphasize-position: after;
fems<Tab>    => font-emphasize-style: none;
fems:n<Tab>  => font-emphasize-style: none;
fems:ac<Tab>=> font-emphasize-style: accent;
fems:dt<Tab>=> font-emphasize-style: dot;
fems:ds<Tab>=> font-emphasize-style: disc;
fems:c<Tab>  => font-emphasize-style: circle;
fsm<Tab>     => font-smoothing: antialiased;
fsm:a<Tab>   => font-smoothing: antialiased;
fsm:s<Tab>   => font-smoothing: subpixel-antialiased;
fsm:n<Tab>   => font-smoothing: none;
fst<Tab>     => font-stretch: normal;
fst:n<Tab>   => font-stretch: normal;
fst:uc<Tab>  => font-stretch: ultra-condensed;
fst:ue<Tab>  => font-stretch: ultra-expanded;
fst:c<Tab>   => font-stretch: condensed;
fst:ec<Tab>  => font-stretch: extra-condensed;
fst:ee<Tab>  => font-stretch: extra-expanded;
fst:e<Tab>   => font-stretch: expanded;
fst:sc<Tab>  => font-stretch: semi-condensed;
fst:se<Tab>  => font-stretch: semi-expanded;
fvs<Tab>     => font-variation-settings: normal;
fvs:n<Tab>   => font-variation-settings: normal;
fvs:u<Tab>   => font-variation-settings: unset;
fvs:ih<Tab>  => font-variation-settings: inherit;
fvs:il<Tab>  => font-variation-settings: initial;

```

1.9.2. Форматирование текста

Расстояние между символами в словах:

```
lts<Tab>    => letter-spacing: normal;  
lts:n<Tab>  => letter-spacing: normal;
```

Расстояние между словами:

```
wos<Tab>    => word-spacing: ;
```

Отступ первой строки:

```
ti<Tab>      => text-indent: ;
```

Вертикальное расстояние между строками:

```
lh<Tab>      => line-height: ;
```

Горизонтальное выравнивание текста:

```
ta<Tab>      => text-align: left;  
ta:l<Tab>    => text-align: left;  
ta:r<Tab>    => text-align: right;  
ta:c<Tab>    => text-align: center;  
ta:j<Tab>    => text-align: justify;  
tal<Tab>     => text-align-last: left;  
tal:l<Tab>   => text-align-last: left;  
tal:r<Tab>   => text-align-last: right;  
tal:c<Tab>   => text-align-last: center;  
tal:a<Tab>   => text-align-last: auto;
```

Вертикальное выравнивание текста:

```
va<Tab>      => vertical-align: top;  
va:t<Tab>    => vertical-align: top;  
va:bl<Tab>   => vertical-align: baseline;  
va:m<Tab>    => vertical-align: middle;  
va:b<Tab>    => vertical-align: bottom;  
va:sub<Tab>  => vertical-align: sub;  
va:sup<Tab>  => vertical-align: super;  
va:tt<Tab>   => vertical-align: text-top;  
va:tb<Tab>   => vertical-align: text-bottom;
```

Подчеркивание, надчеркивание и зачеркивание текста:

```
td<Tab>      => text-decoration: none;  
td:n<Tab>    => text-decoration: none;  
td:u<Tab>    => text-decoration: underline;  
td:o<Tab>    => text-decoration: overline;  
td:l<Tab>    => text-decoration: line-through;
```

Изменение регистра символов:

```
tt<Tab>      => text-transform: uppercase;  
tt:u<Tab>    => text-transform: uppercase;
```

```
tt:l<Tab>    => text-transform: lowercase;
tt:c<Tab>    => text-transform: capitalize;
tt:n<Tab>    => text-transform: none;
```

Обработка пробелов между словами:

```
whs<Tab>     => white-space: nowrap;
whs:nw<Tab>  => white-space: nowrap;
whs:n<Tab>   => white-space: normal;
whs:p<Tab>   => white-space: pre;
whs:pl<Tab>  => white-space: pre-line;
whs:pw<Tab>  => white-space: pre-wrap;
```

Перенос слов:

```
wow<Tab>     => word-wrap: none;
wow:n<Tab>   => word-wrap: none;
wow:nm<Tab>  => word-wrap: normal;
wow:b<Tab>   => word-wrap: break-word;
wow:u<Tab>   => word-wrap: unrestricted;
wow:s<Tab>   => word-wrap: suppress;
wob<Tab>     => word-break: normal;
wob:n<Tab>   => word-break: normal;
wob:b<Tab>   => word-break: break-all;
wob:k<Tab>   => word-break: keep-all;
tw<Tab>      => text-wrap: none;
tw:n<Tab>    => text-wrap: none;
tw:nl<Tab>   => text-wrap: normal;
tw:u<Tab>    => text-wrap: unrestricted;
tw:s<Tab>    => text-wrap: suppress;
```

Атрибут content:

```
cnt<Tab>     => content: '';
ct<Tab>      => content: '';
cnt:n<Tab>   => content: normal;
ct:n<Tab>    => content: normal;
cnt:a<Tab>   => content: attr();
ct:a<Tab>    => content: attr();
cnt:oq<Tab>  => content: open-quote;
ct:oq<Tab>   => content: open-quote;
cnt:cq<Tab>  => content: close-quote;
ct:cq<Tab>   => content: close-quote;
cnt:noq<Tab> => content: no-open-quote;
ct:noq<Tab>  => content: no-open-quote;
cnt:ncq<Tab> => content: no-close-quote;
ct:ncq<Tab>  => content: no-close-quote;
cnt:c<Tab>   => content: counter();
ct:c<Tab>    => content: counter();
cnt:cns<Tab> => content: counters();
```

```
ct:cns<Tab>  => content: counters();
q<Tab>       => quotes: ;
coi<Tab>     => counter-increment: ;
cor<Tab>     => counter-reset: ;
```

Прочие атрибуты:

```
te<Tab>      => text-emphasis: none;
te:n<Tab>    => text-emphasis: none;
te:ac<Tab>   => text-emphasis: accent;
te:dt<Tab>   => text-emphasis: dot;
te:c<Tab>    => text-emphasis: circle;
te:ds<Tab>   => text-emphasis: disc;
te:b<Tab>    => text-emphasis: before;
te:a<Tab>    => text-emphasis: after;
th<Tab>      => text-height: auto;
th:a<Tab>    => text-height: auto;
th:f<Tab>    => text-height: font-size;
th:t<Tab>    => text-height: text-size;
th:m<Tab>    => text-height: max-size;
tj<Tab>      => text-justify: auto;
tj:a<Tab>    => text-justify: auto;
tj:iw<Tab>   => text-justify: inter-word;
tj:ii<Tab>   => text-justify: inter-ideograph;
tj:ic<Tab>   => text-justify: inter-cluster;
tj:d<Tab>    => text-justify: distribute;
tj:k<Tab>    => text-justify: kashida;
tj:t<Tab>    => text-justify: tibetan;
to<Tab>      => text-outline: 0 0 #000;
tr<Tab>      => text-replace: ;
tov<Tab>     => text-overflow: ellipsis;
tov:e<Tab>   => text-overflow: ellipsis;
tov:c<Tab>   => text-overflow: clip;
```

1.9.3. Отступы

Внешние отступы:

```
m<Tab>       => margin: ;
m:a<Tab>     => margin: auto;
m10<Tab>     => margin: 10px;
m10px<Tab>   => margin: 10px;
ml<Tab>      => margin-left: ;
ml:a<Tab>    => margin-left: auto;
mr<Tab>      => margin-right: ;
mr:a<Tab>    => margin-right: auto;
mt<Tab>      => margin-top: ;
mt:a<Tab>    => margin-top: auto;
```

```
mb<Tab>    => margin-bottom: ;
mb:a<Tab>  => margin-bottom: auto;
```

Внутренние отступы:

```
p<Tab>      => padding: ;
p10<Tab>    => padding: 10px;
p10px<Tab>  => padding: 10px;
pl<Tab>     => padding-left: ;
pr<Tab>     => padding-right: ;
pt<Tab>     => padding-top: ;
pb<Tab>     => padding-bottom: ;
```

1.9.4. Рамки

Одновременное задание характеристик рамки:

```
bd<Tab>      => border: 1px solid #000;
bd:n<Tab>    => border: none;
bdl<Tab>     => border-left: 1px solid #000;
bdr<Tab>     => border-right: 1px solid #000;
bdt<Tab>     => border-top: 1px solid #000;
bdb<Tab>     => border-bottom: 1px solid #000;
```

Стиль линии рамки:

```
bds<Tab>     => border-style: none;
bds:n<Tab>   => border-style: none;
bds:h<Tab>   => border-style: hidden;
bds:s<Tab>   => border-style: solid;
bds:dt<Tab>  => border-style: dotted;
bds:ds<Tab>  => border-style: dashed;
bds:db<Tab>  => border-style: double;
bds:g<Tab>   => border-style: groove;
bds:r<Tab>   => border-style: ridge;
bds:i<Tab>   => border-style: inset;
bds:o<Tab>   => border-style: outset;
bds:w<Tab>   => border-style: wave;
bds:dtds<Tab> => border-style: dot-dash;
bds:dttdts<Tab> => border-style: dot-dot-dash;
bdls<Tab>     => border-left-style: ;
bdrst<Tab>    => border-right-style: ;
bdts<Tab>     => border-top-style: ;
bdbbs<Tab>    => border-bottom-style: ;
```

Толщина линии рамки:

```
bdw<Tab>     => border-width: ;
bdlw<Tab>    => border-left-width: ;
bdrw<Tab>    => border-right-width: ;
```

```
bdtw<Tab>    => border-top-width: ;  
bdbw<Tab>    => border-bottom-width: ;
```

Цвет линии рамки:

```
bdc<Tab>      => border-color: #000;  
bdlc<Tab>     => border-left-color: #000;  
bdrc<Tab>     => border-right-color: #000;  
bdtc<Tab>     => border-top-color: #000;  
bdbc<Tab>     => border-bottom-color: #000;
```

Рамки со скругленными углами:

```
bdrs<Tab>     => border-radius: ;  
bdtlrs<Tab>   => border-top-left-radius: ;  
bdtrrs<Tab>   => border-top-right-radius: ;  
bdblrs<Tab>   => border-bottom-left-radius: ;  
bdbrrs<Tab>   => border-bottom-right-radius: ;
```

Атрибут border-image:

```
bdi<Tab>      => border-image: url();  
bdli<Tab>     => border-left-image: url();  
bdri<Tab>     => border-right-image: url();  
bdti<Tab>     => border-top-image: url();  
bdbi<Tab>     => border-bottom-image: url();  
bdtli<Tab>    => border-top-left-image: url();  
bdtli:c<Tab>  => border-top-left-image: continue;  
bdtri<Tab>    => border-top-right-image: url();  
bdtri:c<Tab>  => border-top-right-image: continue;  
bdbli<Tab>    => border-bottom-left-image: url();  
bdbli:c<Tab>  => border-bottom-left-image: continue;  
bdbri<Tab>    => border-bottom-right-image: url();  
bdbri:c<Tab>  => border-bottom-right-image: continue;
```

Внешняя рамка:

```
ol<Tab>       => outline: ;  
ols<Tab>      => outline-style: none;  
ols:n<Tab>    => outline-style: none;  
ols:s<Tab>    => outline-style: solid;  
ols:dt<Tab>   => outline-style: dotted;  
ols:ds<Tab>   => outline-style: dashed;  
ols:db<Tab>   => outline-style: double;  
ols:g<Tab>    => outline-style: groove;  
ols:r<Tab>    => outline-style: ridge;  
ols:i<Tab>    => outline-style: inset;  
ols:o<Tab>    => outline-style: outset;  
olc<Tab>      => outline-color: #000;  
olc:i<Tab>    => outline-color: invert;  
olw<Tab>      => outline-width|thin|medium|thick  
olo<Tab>      => outline-offset: ;
```


Прочие атрибуты:

```
bdbk<Tab>    => border-break: close;
bdbk:c<Tab>   => border-break: close;
bdci<Tab>     => border-corner-image: url();
bdci:c<Tab>   => border-corner-image: continue;
bdf<Tab>      => border-fit: repeat;
bdf:r<Tab>    => border-fit: repeat;
bdf:c<Tab>    => border-fit: clip;
bdf:sc<Tab>   => border-fit: scale;
bdf:st<Tab>   => border-fit: stretch;
bdf:ow<Tab>   => border-fit: overwrite;
bdf:of<Tab>   => border-fit: overflow;
bdf:sp<Tab>   => border-fit: space;
bdlen<Tab>    => border-length: ;
bdlen:a<Tab>  => border-length: auto;
```

1.9.5. Фон элемента

Одновременное задание характеристик фона:

```
bg<Tab>      => background: #000;
```

Цвет фона:

```
bgc<Tab>     => background-color: #fff;
```

Фоновый рисунок:

```
bgi<Tab>     => background-image: url();
lg<Tab>      => background-image: linear-gradient();
```

Режим повтора фонового рисунка:

```
bgr<Tab>     => background-repeat: no-repeat;
bgr:n<Tab>   => background-repeat: no-repeat;
bgr:rx<Tab>  => background-repeat: repeat-x;
bgr:ry<Tab>  => background-repeat: repeat-y;
bgr:rd<Tab>  => background-repeat: round;
bgr:sp<Tab>  => background-repeat: space;
```

Прокрутка фонового рисунка:

```
bga<Tab>     => background-attachment: fixed;
bga:f<Tab>   => background-attachment: fixed;
bga:s<Tab>   => background-attachment: scroll;
```

Положение фонового рисунка:

```
bgp<Tab>     => background-position: 0 0;
bgpx<Tab>    => background-position-x: ;
bgpy<Tab>    => background-position-y: ;
```

Размеры фонового изображения:

```
bgsz<Tab>    => background-size: contain;  
bgsz:ct<Tab> => background-size: contain;  
bgsz:a<Tab>  => background-size: auto;  
bgsz:cv<Tab> => background-size: cover;
```

Режим позиционирования фонового изображения:

```
bgo<Tab>      => background-origin: padding-box;  
bgo:pb<Tab>   => background-origin: padding-box;  
bgo:bb<Tab>   => background-origin: border-box;  
bgo:cb<Tab>   => background-origin: content-box;
```

Режим заполнения для фона:

```
bgcp<Tab>     => background-clip: padding-box;  
bgcp:pb<Tab>  => background-clip: padding-box;  
bgcp:cb<Tab>  => background-clip: content-box;  
bgcp:bb<Tab>  => background-clip: border-box;  
bgcp:nc<Tab>  => background-clip: no-clip;
```

Атрибут background-break:

```
bgbk<Tab>     => background-break: bounding-box;  
bgbk:bb<Tab>  => background-break: bounding-box;  
bgbk:eb<Tab>  => background-break: each-box;  
bgbk:c<Tab>   => background-break: continuous;
```

1.9.6. Списки

Одновременное указание характеристик списка:

```
lis<Tab>      => list-style: ;
```

Вид маркера списка:

```
list<Tab>     => list-style-type: disc;  
list:d<Tab>   => list-style-type: disc;  
list:c<Tab>   => list-style-type: circle;  
list:s<Tab>   => list-style-type: square;  
list:dl<Tab>  => list-style-type: decimal;  
list:dlz<Tab> => list-style-type: decimal-leading-zero;  
list:lr<Tab>  => list-style-type: lower-roman;  
list:ur<Tab>  => list-style-type: upper-roman;
```

Изображение в качестве маркера списка:

```
lisi<Tab>     => list-style-image: ;
```

Компактное отображение списка:

```
lisp<Tab>     => list-style-position: inside;  
lisp:i<Tab>   => list-style-position: inside;  
lisp:o<Tab>   => list-style-position: outside;
```

1.9.7. Таблицы

Рамки таблицы и ячеек:

```
bdsp<Tab>    => border-spacing: ;  
bdcl<Tab>    => border-collapse: collapse;  
bdcl:c<Tab>  => border-collapse: collapse;  
bdcl:s<Tab>  => border-collapse: separate;
```

Размеры таблицы:

```
tbl<Tab>     => table-layout: fixed;  
tbl:f<Tab>   => table-layout: fixed;  
tbl:a<Tab>   => table-layout: auto;
```

Местоположение заголовка:

```
cps<Tab>     => caption-side: top;  
cps:t<Tab>   => caption-side: top;  
cps:b<Tab>   => caption-side: bottom;
```

Указание характеристик ячеек:

```
ec<Tab>      => empty-cells: show;  
ec:s<Tab>    => empty-cells: show;  
ec:h<Tab>    => empty-cells: hide;
```

1.9.8. Вид курсора

Вид курсора:

```
cur<Tab>     => cursor: pointer;  
cur:p<Tab>   => cursor: pointer;  
cur:a<Tab>   => cursor: auto;  
cur:c<Tab>   => cursor: crosshair;  
cur:d<Tab>   => cursor: default;  
cur:m<Tab>   => cursor: move;  
cur:t<Tab>   => cursor: text;  
cur:he<Tab> => cursor: help;  
cur:ha<Tab> => cursor: hand;
```

1.9.9. Форматирование блоков

Управление отображением:

```
v<Tab>       => visibility: hidden;  
v:h<Tab>     => visibility: hidden;  
v:v<Tab>     => visibility: visible;  
v:c<Tab>     => visibility: collapse;
```

Указание типа блока:

```
d<Tab>       => display: grid;  
d:n<Tab>     => display: none;
```

```
d:b<Tab>      => display: block;
d:i<Tab>      => display: inline;
d:ib<Tab>     => display: inline-block;
d:li<Tab>     => display: list-item;
d:tb<Tab>     => display: table;
d:itb<Tab>    => display: inline-table;
d:tbc<Tab>    => display: table-caption;
d:tbg<Tab>    => display: table-header-group;
d:tbrg<Tab>   => display: table-row-group;
d:tbf<Tab>    => display: table-footer-group;
d:tbr<Tab>    => display: table-row;
d:tbc<Tab>    => display: table-column;
d:tbcg<Tab>   => display: table-column-group;
d:tbc<Tab>    => display: table-cell;
d:f<Tab>      => display: flex;
d:if<Tab>     => display: inline-flex;
d:g<Tab>      => display: grid;
d:ig<Tab>     => display: inline-grid;
d:s<Tab>      => display: subgrid;
d:ri<Tab>     => display: run-in;
d:cp<Tab>     => display: compact;
d:rb<Tab>     => display: ruby;
d:r<Tab>      => display: ruby-text;
d:r<Tab>      => display: ruby-text-group;
```

Указание размеров:

```
bxs<Tab>      => box-sizing: border-box;
bxs:bb<Tab>   => box-sizing: border-box;
bxs:cb<Tab>   => box-sizing: content-box;
w<Tab>        => width: ;
w:a<Tab>      => width: auto;
h<Tab>        => height: ;
h:a<Tab>      => height: auto;
miw<Tab>      => min-width: ;
mih<Tab>      => min-height: ;
maw<Tab>      => max-width: ;
mah<Tab>      => max-height: ;
```

Атрибут overflow:

```
ov<Tab>       => overflow: hidden;
ov:h<Tab>     => overflow: hidden;
ov:v<Tab>     => overflow: visible;
ov:s<Tab>     => overflow: scroll;
ov:a<Tab>     => overflow: auto;
ovx<Tab>     => overflow-x: hidden;
ovx:h<Tab>    => overflow-x: hidden;
ovx:v<Tab>    => overflow-x: visible;
ovx:s<Tab>    => overflow-x: scroll;
```

```

ovx:a<Tab>  => overflow-x: auto;
ovy<Tab>    => overflow-y: hidden;
ovy:h<Tab>  => overflow-y: hidden;
ovy:v<Tab>  => overflow-y: visible;
ovy:s<Tab>  => overflow-y: scroll;
ovy:a<Tab>  => overflow-y: auto;
ovs<Tab>    => overflow-style: scrollbar;
ovs:s<Tab>  => overflow-style: scrollbar;
ovs:a<Tab>  => overflow-style: auto;
ovs:p<Tab>  => overflow-style: panner;
ovs:m<Tab>  => overflow-style: move;
ovs:mq<Tab> => overflow-style: marquee;

```

Управление обтеканием:

```

fl<Tab>     => float: left;
fl:l<Tab>   => float: left;
fl:r<Tab>   => float: right;
fl:n<Tab>   => float: none;
cl<Tab>     => clear: both;
cl:b<Tab>   => clear: both;
cl:l<Tab>   => clear: left;
cl:r<Tab>   => clear: right;
cl:n<Tab>   => clear: none;

```

Позиционирование блока:

```

pos<Tab>    => position: relative;
pos:r<Tab>  => position: relative;
pos:a<Tab>  => position: absolute;
pos:f<Tab>  => position: fixed;
pos:s<Tab>  => position: static;
l<Tab>      => left: ;
l:a<Tab>    => left: auto;
t<Tab>      => top: ;
t:a<Tab>    => top: auto;
r<Tab>      => right: ;
r:a<Tab>    => right: auto;
b<Tab>      => bottom: ;
b:a<Tab>    => bottom: auto;

```

Последовательность отображения слоев:

```

z<Tab>      => z-index: ;
z:a<Tab>    => z-index: auto;

```

1.9.10. Flex-контейнеры

Указание типа блока:

```

d:f<Tab>    => display: flex;
d:if<Tab>   => display: inline-flex;

```

Направление выравнивания элементов внутри контейнера:

```
fxd<Tab>    => flex-direction: row;
fxd:r<Tab>   => flex-direction: row;
fxd:rr<Tab>  => flex-direction: row-reverse;
fxd:c<Tab>   => flex-direction: column;
fxd:cr<Tab>  => flex-direction: column-reverse;
```

Перенос на новую строку:

```
fxw<Tab>    => flex-wrap: nowrap;
fxw:n<Tab>   => flex-wrap: nowrap;
fxw:w<Tab>   => flex-wrap: wrap;
fxw:wv<Tab>  => flex-wrap: wrap-reverse;
fxw:we<Tab>  => flex-wrap: wrap-reverse;
```

Одновременное указание характеристик флекс-контейнера:

```
fxf<Tab>    => flex-flow: ;
```

Размеры элемента:

```
fxb<Tab>     => flex-basis: fill;
fxb:f<Tab>    => flex-basis: fill;
fxb:a<Tab>    => flex-basis: auto;
fxb:mic<Tab>  => flex-basis: min-content;
fxb:mac<Tab>  => flex-basis: max-content;
fxb:fc<Tab>   => flex-basis: fit-content;
fxb:c<Tab>    => flex-basis: content;
```

Растяжение элементов:

```
fxg<Tab>     => flex-grow: ;
```

Сжатие элементов:

```
fxsh<Tab>    => flex-shrink: ;
```

Одновременное указание характеристик элементов:

```
fx<Tab>      => flex: ;
```

Выравнивание элементов внутри контейнера:

```
jc<Tab>      => justify-content: ;
jc:fs<Tab>   => justify-content: flex-start;
jc:fe<Tab>   => justify-content: flex-end;
jc:c<Tab>    => justify-content: center;
jc:sb<Tab>   => justify-content: space-between;
jc:sd<Tab>   => justify-content: space-around;
jc:sy<Tab>   => justify-content: space-evenly;
ac<Tab>      => align-content: ;
ac:sh<Tab>   => align-content: stretch;
ac:fs<Tab>   => align-content: flex-start;
ac:fe<Tab>   => align-content: flex-end;
```

```

ac:c<Tab>    => align-content: center;
ac:sb<Tab>   => align-content: space-between;
ac:sd<Tab>   => align-content: space-around;
ac:sy<Tab>   => align-content: space-evenly;
ai<Tab>      => align-items: ;
ai:sh<Tab>   => align-items: stretch;
ai:fs<Tab>   => align-items: flex-start;
ai:fe<Tab>   => align-items: flex-end;
ai:c<Tab>    => align-items: center;
ai:b<Tab>    => align-items: baseline;
as<Tab>      => align-self: ;
as:a<Tab>    => align-self: auto;
as:sh<Tab>   => align-self: stretch;
as:fs<Tab>   => align-self: flex-start;
as:fe<Tab>   => align-self: flex-end;
as:c<Tab>    => align-self: center;
as:b<Tab>    => align-self: baseline;

```

Порядок следования элементов внутри контейнера:

```
ord<Tab>    => order: ;
```

1.9.11. CSS Grid

Указание типа блока:

```

d<Tab>      => display: grid;
d:g<Tab>    => display: grid;
d:ig<Tab>   => display: inline-grid;
d:s<Tab>    => display: subgrid;

```

Описание строк и столбцов:

```

gt<Tab>     => grid-template: ;
gtc<Tab>    => grid-template-columns: repeat();
gtc:r<Tab>  => grid-template-columns: repeat();
gtc:m<Tab>  => grid-template-columns: minmax();
gtc:a<Tab>  => grid-template-columns: auto;
gtr<Tab>    => grid-template-rows: repeat();
gtr:m<Tab>  => grid-template-rows: minmax();
gtr:a<Tab>  => grid-template-rows: auto;

```

Автоматическое размещение элементов внутри сетки:

```

gaf<Tab>    => grid-auto-flow: row;
gaf:r<Tab>  => grid-auto-flow: row;
gaf:c<Tab>  => grid-auto-flow: column;
gaf:d<Tab>  => grid-auto-flow: dense;
gaf:i<Tab>  => grid-auto-flow: initial;
gaf:ih<Tab> => grid-auto-flow: inherit;
gaf:u<Tab>  => grid-auto-flow: unset;

```

Добавление элементов в ячейки сетки:

```
gcs<Tab>  => grid-column-start: ;
gce<Tab>  => grid-column-end: ;
gc<Tab>   => grid-column: ;
grs<Tab>  => grid-row-start: ;
gre<Tab>  => grid-row-end: ;
gr<Tab>   => grid-row: ;
ga<Tab>   => grid-area: ;
```

Размеры неявных ячеек:

```
gac<Tab>  => grid-auto-columns: auto;
gac:m<Tab>=> grid-auto-columns: minmax();
gar<Tab>  => grid-auto-rows: auto;
gar:m<Tab>=> grid-auto-rows: minmax();
```

Расстояние между ячейками:

```
gcg<Tab>  => grid-column-gap: ;
grg<Tab>  => grid-row-gap: ;
gg<Tab>   => grid-gap: ;
```

Имена элементов:

```
ga<Tab>  => grid-area: ;
gta<Tab> => grid-template-areas: ;
gt<Tab>  => grid-template: ;
```

Одновременное указание характеристик контейнера:

```
gd<Tab>  => grid: ;
```

Выравнивание сетки внутри контейнера:

```
jc<Tab>  => justify-content: start;
jc:s<Tab>=> justify-content: start;
jc:e<Tab>=> justify-content: end;
jc:c<Tab>=> justify-content: center;
jc:sb<Tab>=> justify-content: space-between;
jc:sd<Tab>=> justify-content: space-around;
jc:sy<Tab>=> justify-content: space-evenly;
ac<Tab>  => align-content: start;
ac:s<Tab>=> align-content: start;
ac:e<Tab>=> align-content: end;
ac:c<Tab>=> align-content: center;
ac:sb<Tab>=> align-content: space-between;
ac:sd<Tab>=> align-content: space-around;
ac:sy<Tab>=> align-content: space-evenly;
```

Выравнивание элемента внутри ячейки:

```
js<Tab>  => justify-self: start;
js:s<Tab>=> justify-self: start;
```



```
js:e<Tab>    => justify-self: end;
js:c<Tab>    => justify-self: center;
js:sh<Tab>   => justify-self: stretch;
ji<Tab>      => justify-items: start;
ji:s<Tab>    => justify-items: start;
ji:e<Tab>    => justify-items: end;
ji:c<Tab>    => justify-items: center;
ji:sh<Tab>   => justify-items: stretch;
as<Tab>      => align-self: start;
as:s<Tab>    => align-self: start;
as:e<Tab>    => align-self: end;
as:sh<Tab>   => align-self: stretch;
as:c<Tab>    => align-self: center;
as:a<Tab>    => align-self: auto;
ai<Tab>      => align-items: start;
ai:s<Tab>    => align-items: start;
ai:e<Tab>    => align-items: end;
ai:sh<Tab>   => align-items: stretch;
ai:c<Tab>    => align-items: center;
```

1.9.12. Многоколоночный текст

Число колонок:

```
colm<Tab>    => columns: ;
colmc<Tab>   => column-count: ;
colmc:a<Tab> => column-count: auto;
```

Размеры колонок:

```
colmw<Tab>    => column-width: ;
colmw:a<Tab>  => column-width: auto;
colmf<Tab>    => column-fill: ;
colmf:a<Tab>  => column-fill: auto;
```

Расстояние между колонками:

```
colmg<Tab>    => column-gap: ;
```

Линии между колонками:

```
colmr<Tab>    => column-rule: ;
colmrs<Tab>   => column-rule-style: ;
colmrc<Tab>   => column-rule-color: ;
colmrw<Tab>   => column-rule-width: ;
colms<Tab>    => column-span: ;
```

1.9.13. Эффекты

Изменение прозрачности:

```
op<Tab>       => opacity: ;
```

Создание тени для текста:

```
tsh<Tab>    => text-shadow: hoff voff blur #000;
```

Создание тени для блока:

```
bxsh<Tab>    => box-shadow: inset hoff voff blur #000;
```

```
bxsh:n<Tab> => box-shadow: none;
```

1.9.14. Анимация с двумя состояниями

Одновременное задание всех параметров анимации:

```
trs<Tab>     => transition: prop time;
```

Продолжительность анимации:

```
trsd<Tab>    => transition-duration: time;
```

Задержка перед началом анимации:

```
trsd<Tab>    => transition-delay: time;
```

Задание анимируемых атрибутов:

```
trsp<Tab>    => transition-property: prop;
```

Закон анимации:

```
trstf<Tab>   => transition-timing-function: fn;
```

1.9.15. Анимация с несколькими состояниями

Шкала времени:

```
@kf<Tab>     =>
  @keyframes identifier {
  }
```

Одновременное задание всех параметров анимации:

```
anim<Tab>     =>
  animation: name duration timing-function delay iteration-count ↵
             direction fill-mode;
```

Указание имени шкалы времени:

```
animn<Tab>    => animation-name: ;
```

Продолжительность анимации:

```
animdur<Tab>  => animation-duration: 0s;
```

Задержка перед началом анимации:

```
animdel<Tab>  => animation-delay: time;
```

Закон анимации:

```
animtf<Tab>      => animation-timing-function: linear;
animtf:l<Tab>    => animation-timing-function: linear;
animtf:e<Tab>    => animation-timing-function: ease;
animtf:ei<Tab>   => animation-timing-function: ease-in;
animtf:eo<Tab>   => animation-timing-function: ease-out;
animtf:eio<Tab>  => animation-timing-function: ease-in-out;
animtf:cb<Tab>   =>
    animation-timing-function: cubic-bezier(0.1, 0.7, 1.0, 1.0);
```

Количество повторений анимации:

```
animic<Tab>      => animation-iteration-count: 1;
animic:i<Tab>    => animation-iteration-count: infinite;
```

Направление анимации:

```
animdir<Tab>     => animation-direction: normal;
animdir:n<Tab>   => animation-direction: normal;
animdir:r<Tab>   => animation-direction: reverse;
animdir:a<Tab>   => animation-direction: alternate;
animdir:arv<Tab>=> animation-direction: alternate-reverse;
```

Текущее состояние анимации:

```
animps<Tab>      => animation-play-state: running;
animps:r<Tab>    => animation-play-state: running;
animps:p<Tab>    => animation-play-state: paused;
```

Состояние элемента до начала анимации и после ее завершения:

```
animfm<Tab>      => animation-fill-mode: both;
animfm:bh<Tab>   => animation-fill-mode: both;
animfm:bt<Tab>   => animation-fill-mode: both;
animfm:bs<Tab>   => animation-fill-mode: backwards;
animfm:f<Tab>    => animation-fill-mode: forwards;
```

1.9.16. Двумерные трансформации

Атрибут transform:

```
trf<Tab>         => transform: ;
```

Смещение:

```
trf:tx<Tab>      => transform: translateX(x);
trf:ty<Tab>      => transform: translateY(y);
trf:t<Tab>       => transform: translate(x, y);
```

Изменение масштаба:

```
trf:scx<Tab>     => transform: scaleX(x);
trf:scy<Tab>     => transform: scaleY(y);
trf:sc<Tab>      => transform: scale(x, y);
```

Наклон:

```
trf:skx<Tab>      => transform: skewX(angle);
trf:sky<Tab>      => transform: skewY(angle);
```

Вращение:

```
trf:r<Tab>        => transform: rotate(angle);
```

Позиционирование точки начала координат для двумерных трансформаций:

```
trfo<Tab>         => transform-origin: ;
```

1.9.17. Трехмерные трансформации**Выполнение трехмерных трансформаций:**

```
trf:tz<Tab>       => transform: translateZ(z);
trf:td<Tab>       => transform: translate3d(tx, ty, tz);
trf:scz<Tab>      => transform: scaleZ(z);
trf:scd<Tab>      => transform: scale3d(x, y, z);
trf:rx<Tab>       => transform: rotateX(angle);
trf:ry<Tab>       => transform: rotateY(angle);
trf:rz<Tab>       => transform: rotateZ(angle);
```

Скрытие обратной стороны элемента:

```
bfv<Tab>          => backface-visibility: hidden;
bfv:h<Tab>        => backface-visibility: hidden;
bfv:v<Tab>        => backface-visibility: visible;
```

Режим проецирования элементов на контейнер:

```
trfs<Tab>         => transform-style: preserve-3d;
```

Позиционирование точки начала координат для трехмерных трансформаций:

```
trfo<Tab>         => transform-origin: ;
```

1.9.18. Прочие атрибуты и правила**Правила:**

```
@i<Tab>           => @import url();
@import<Tab>      => @import url();
@m<Tab>           =>
    @media screen {

    }
@media<Tab>      =>
    @media screen {

    }
```

Прочие атрибуты:

```

ap<Tab>      => appearance: none;
cp<Tab>      => clip: auto;
cp:a<Tab>    => clip: auto;
cp:r<Tab>    => clip: rect(top right bottom left);
mar<Tab>     => max-resolution: ;
mir<Tab>     => min-resolution: ;
ori<Tab>     => orientation: landscape;
ori:l<Tab>   => orientation: landscape;
ori:p<Tab>   => orientation: portrait;
orp<Tab>     => orphans: ;
pgbb<Tab>    => page-break-before: auto;
pgbb:au<Tab> => page-break-before: auto;
pgbb:al<Tab> => page-break-before: always;
pgbb:l<Tab>  => page-break-before: left;
pgbb:r<Tab>  => page-break-before: right;
pgba<Tab>    => page-break-after: auto;
pgba:au<Tab> => page-break-after: auto;
pgba:al<Tab> => page-break-after: always;
pgba:l<Tab>  => page-break-after: left;
pgba:r<Tab>  => page-break-after: right;
pgbi<Tab>    => page-break-inside: auto;
pgbi:au<Tab> => page-break-inside: auto;
pgbi:av<Tab> => page-break-inside: avoid;
rsz<Tab>     => resize: none;
rsz:n<Tab>   => resize: none;
rsz:b<Tab>   => resize: both;
rsz:h<Tab>   => resize: horizontal;
rsz:v<Tab>   => resize: vertical;
whsc<Tab>    => white-space-collapse: normal;
whsc:n<Tab>  => white-space-collapse: normal;
whsc:k<Tab>  => white-space-collapse: keep-all;
whsc:l<Tab>  => white-space-collapse: loose;
whsc:bs<Tab> => white-space-collapse: break-strict;
whsc:ba<Tab> => white-space-collapse: break-all;
wm<Tab>      => writing-mode: lr-tb;
wm:bt<Tab>   => writing-mode: bt-lr;
wm:btr<Tab>  => writing-mode: bt-rl;
wm:lrb<Tab>  => writing-mode: lr-bt;
wm:lrt<Tab>  => writing-mode: lr-tb;
wm:rlb<Tab>  => writing-mode: rl-bt;
wm:rlt<Tab>  => writing-mode: rl-tb;
wm:tbt<Tab>  => writing-mode: tb-lr;
wm:tbr<Tab>  => writing-mode: tb-rl;
us<Tab>      => user-select: none;
zom<Tab>     => zoom: 1;

```

1.9.19. Ввод числового значения атрибута

После комбинации символов допускается указание числового значения атрибута стилей. Emmet при необходимости сам добавит к нему единицу изменения `px` (у целочисленных значений) или `em` (у дробных значений). Пример:

```
m10<Tab>    => margin: 10px;
p2.5<Tab>    => padding: 2.5em;
lh1.5<Tab>   => line-height: 1.5;
```

Если нужно указать несколько значений, то они задаются через дефис:

```
m10-20<Tab> => margin: 10px 20px;
```

Для ввода отрицательных значений следует указать два дефиса:

```
p-15--5<Tab> => padding: -15px -5px;
```

После значения можно указать единицу измерения. В этом случае разделять значения дефисом не нужно. Пример:

```
p10px20px<Tab> => padding: 10px 20px;
```

Вместо реальной единицы измерения допускается указание псевдонимов: `p` для `%`, `e` для `em`, `x` для `ex`:

```
p10p20e30x5<Tab> => padding: 10% 20em 30ex 5px;
```

Для вставки значения `!important` достаточно ввести восклицательный знак:

```
m20!<Tab>      => margin: 20px !important;
```

В качестве значения цвета можно указать одну, две, три или шесть шестнадцатеричных цифр после символа `#`. Если цифры не заданы, то получим черный цвет. Пример:

```
c#<Tab>        => color: #000000;
c#f<Tab>        => color: #ffffff;
c#e8<Tab>       => color: #e8e8e8;
c#ec0<Tab>      => color: #eecc00;
c#28a745<Tab>   => color: #28a745;
```

1.9.20. Добавление вендорных префиксов

Если перед комбинацией символов указать дефис, то будут сгенерированы имена атрибутов с вендорными префиксами:

```
-bdrs<Tab> =>
  -webkit-border-radius: ;
  -moz-border-radius: ;
  border-radius: ;
```

НА ЗАМЕТКУ

Пакет Autoprefixer позволяет автоматически добавлять вендорные префиксы в зависимости от выбранных веб-браузеров. Поэтому его использование является более правильным решением, чем генерация всех префиксов.

1.10. Всплывающие подсказки при вводе

При вводе первых букв очередной комбинации символов, применяемых в Emmet, VS Code выведет всплывающую подсказку со списком всех вариантов, включающих набранные символы (рис. 1.26). Нужный вариант можно выбрать щелчком мыши или клавишами-стрелками, после чего нажать клавишу <Enter>. Для атрибутов CSS, имена которых состоят из нескольких слов, вывести такую подсказку можно, набрав только первые буквы этих слов.

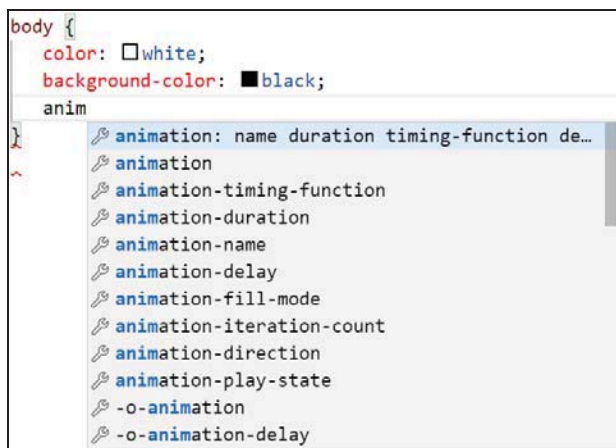


Рис. 1.26. Всплывающая подсказка с совпадениями при вводе букв в CSS-файле

Помимо комбинаций символов Emmet, редактор VS Code предлагает и свои собственные комбинации, зависящие от типа файла. Например, при вводе открывающей угловой скобки в HTML-файле редактор отобразит список всех тегов (рис. 1.27). По мере дальнейшего ввода список совпадений будет уменьшаться. Если в JavaScript-файле ввести имя объекта и добавить точку, будет выведен список всех свойств и методов этого объекта (рис. 1.28).

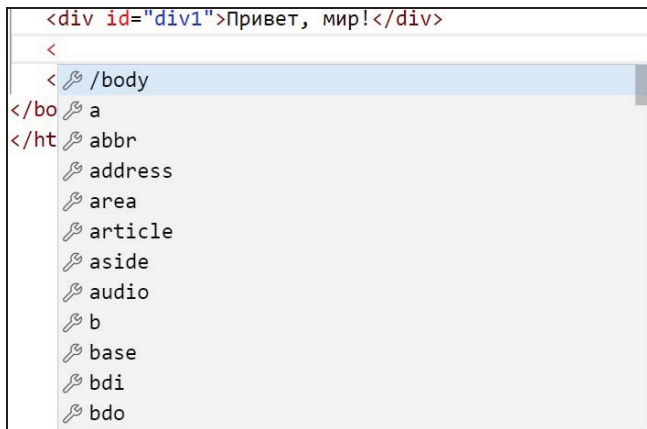


Рис. 1.27. Всплывающая подсказка с именами HTML-тегов



Рис. 1.28. Всплывающая подсказка со списком свойств и методов объекта в JavaScript-файле

Если вы случайно закрыли всплывающую подсказку, например щелкнув мышью вне окна, то для ее повторного вывода достаточно нажать комбинацию клавиш `<Ctrl>+<Пробел>`. Если эту комбинацию клавиш нажать при открытом всплывающем окне, то отобразится еще одна подсказка с дополнительной информацией, например с форматом метода в JavaScript.

1.11. Пользовательские фрагменты кода

Помимо комбинаций символов Emmet и встроенных комбинаций, редактор VS Code отображает во всплывающих подсказках и пользовательские фрагменты кода. Добавить их можно как в Emmet, так и в сам редактор.

1.11.1. В Emmet

Для Emmet нужно указать каталог, в котором мы будем хранить пользовательские фрагменты кода. Создадим каталог `emmet` в каталоге `C:\book`. Затем в настройках редактора для пользователя выполняем поиск настройки **Emmet: Extensions Path**. Нажимаем кнопку **Добавить элемент**, вводим в появившееся поле ввода путь к каталогу `C:\book\emmet` и нажимаем кнопку **ОК**.

Теперь в каталоге `C:\book\emmet` создаем файл `snippets.json` с кодом из листинга 1.4.

Листинг 1.4. Содержимое файла `snippets.json`

```
{
  "html": {
    "snippets": {
      "ul1": "ul>li*3",
      "tbl1": "table>tr*2>td*2"
    }
  },
}
```



```

"css": {
  "snippets": {
    "cmt": "/* ${1} */"
  }
}
}

```

Мы добавили в файл `snippets.json` две секции: `html` и `css`. Внутри каждой секции существует свойство `snippets`, хранящее объект с пользовательскими фрагментами кода. Для HTML-файлов мы создали две пользовательские комбинации:

```

ull<Tab> =>
  <ul>
    <li></li>
    <li></li>
    <li></li>
  </ul>
tbl<Tab> =>
  <table>
    <tr>
      <td></td>
      <td></td>
    </tr>
    <tr>
      <td></td>
      <td></td>
    </tr>
  </table>

```

Для CSS-файлов мы добавили комбинацию вставки комментария:

```
cmt<Tab> => /* */
```

Чтобы после вставки фрагмента курсор находился в нужном месте, в составе значения следует указать символ `$`, после которого внутри фигурных скобок передать номер курсора (нумерация начинается с 1):

```
"cmt": "/* ${1} */"
```

Результат:

```
cmt<Tab> => /* {Курсор} */
```

Можно пометить сразу несколько положений курсора, и тогда при каждом нажатии клавиши `<Tab>` курсор будет обходить все отмеченные места:

```
"cmt": "/* ${1} */\n${2}"
```

После номера можно вставить двоеточие и добавить значение по умолчанию, которое будет выделено после вставки фрагмента:

```
"cmt": "/* ${1:Комментарий по умолчанию} */"
```

Результат:

```
cmt<Tab> => /* Комментарий по умолчанию */
```

После значения внутри секции `css` не следует указывать точку с запятой. Она добавляется автоматически в случае необходимости, в зависимости от типа файла. Например, в SCSS-файле точка с запятой будет добавлена, а в SASS-файле — нет.

После указания пути к каталогу с пользовательскими фрагментами кода в файл `settings.json` будет добавлена следующая настройка:

```
"emmet.extensionsPath": [  
  "C:\\book\\emmet"  
]
```

В настройке **Emmet: Extensions Path** можно указать произвольное количество путей к разным каталогам с пользовательскими фрагментами кода. Emmet будет искать фрагменты кода во всех указанных каталогах.

1.11.2. В редакторе VS Code

Чтобы добавить пользовательскую комбинацию символов в сам редактор VS Code, следует в меню **Файл** выбрать пункт **Настройки | Пользовательские фрагменты кода**. Отобразится панель **Палитра команд** (рис. 1.29). Из списка нужно выбрать тип файла или создать файл с глобальным фрагментом кода для пользователя либо для рабочей области.

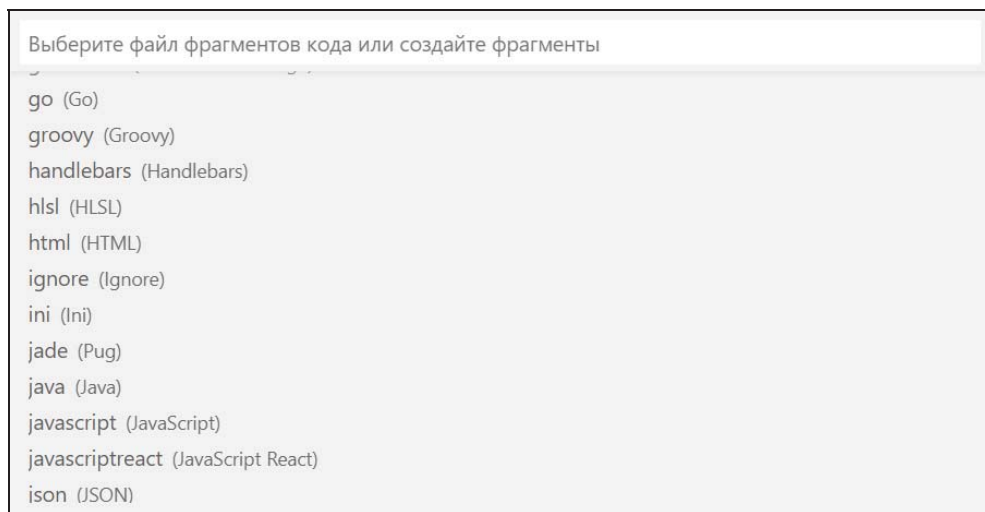


Рис. 1.29. Палитра команд

Выберем тип `html`. В отдельной вкладке откроется содержимое файла `html.json`, который расположен в каталоге `C:\Users\<Имя пользователя>\AppData\Roaming\Code\User\snippets`. Внутри файла уже вставлен закомментированный пример создания пользовательского фрагмента кода. Удаляем все содержимое файла и вставляем код из листинга 1.5.

Листинг 1.5. Содержимое файла html.json

```
{
  "Bootstrap CSS": {
    "prefix": "bscss",
    "body": [
      "<link rel=\"stylesheet\" type=\"text/css\" ↵
      href=\"css/bootstrap.min.css\">",
      "$0"
    ],
    "description": "Bootstrap CSS"
  },
  "Bootstrap JS": {
    "prefix": "bsjs",
    "body": [
      "<script src=\"js/bootstrap.bundle.min.js\"></script>",
      "<script>",
      "</script>",
      "$0"
    ],
    "description": "Bootstrap JS"
  },
  "jQuery JS": {
    "prefix": "jqjs",
    "body": [
      "<script src=\"js/jquery.min.js\"></script>",
      "$0"
    ],
    "description": "jQuery JS"
  }
}
```

В этом примере мы создали комбинации символов для подключения файлов библиотек Bootstrap и jQuery. Название фрагмента кода указывается перед описанием всего блока. В свойстве `description` можно указать описание фрагмента кода. Свойство `prefix` задает комбинацию символов, а свойство `body` — вставляемый фрагмент кода в виде массива строк.

Чтобы после вставки фрагмента курсор находился в нужном месте, в составе значения следует указать символ `$`, после которого передать номер курсора (нумерация начинается с 1). Можно пометить сразу несколько положений курсора, и при каждом нажатии клавиши `<Tab>` курсор будет обходить все отмеченные места. Конечная позиция курсора задается с помощью комбинации `$0`:

```
"body": [
  "<input type=\"$1\" name=\"$2\" value=\"$3\">",
  "$0"
]
```

После номера можно вставить двоеточие и добавить значение по умолчанию, которое будет выделено после вставки фрагмента. Эту комбинацию следует заключить в фигурные скобки. Пример:

```
"<input type=\"${1:text}\" name=\"$2\" value=\"$3\">"
```

Результат:

```
<input type="text" name="" value="">
```

Чтобы пользователь мог выбрать один вариант из нескольких доступных, нужно перечислить доступные варианты через запятую между символами вертикальной черты:

```
"<input type=\"${1|text,password,email|}\" name=\"$2\" value=\"$3\">"
```

В составе строки с фрагментом могут быть указаны следующие переменные:

- ☐ `TM_SELECTED_TEXT` — текущий выделенный текст или пустая строка, если текст не был выделен;
- ☐ `TM_CURRENT_LINE` — содержимое текущей строки;
- ☐ `TM_CURRENT_WORD` — слово под курсором или пустая строка;
- ☐ `TM_LINE_INDEX` — индекс текущей строки (нумерация начинается с 0);
- ☐ `TM_LINE_NUMBER` — номер текущей строки (нумерация начинается с 1);
- ☐ `TM_FILENAME` — имя файла текущего документа с расширением;
- ☐ `TM_FILENAME_BASE` — имя файла текущего документа без расширения;
- ☐ `TM_DIRECTORY` — путь к каталогу, в котором находится текущий документ;
- ☐ `TM_FILEPATH` — путь к файлу текущего документа;
- ☐ `RELATIVE_FILEPATH` — путь к файлу текущего документа относительно каталога, открытого в данный момент в редакторе;
- ☐ `CLIPBOARD` — содержимое буфера обмена;
- ☐ `WORKSPACE_NAME` — имя каталога или рабочей области;
- ☐ `WORKSPACE_FOLDER` — путь к каталогу, открытому в данный момент в редакторе;
- ☐ `CURRENT_YEAR` — текущий год из четырех цифр;
- ☐ `CURRENT_YEAR_SHORT` — текущий год из двух последних цифр;
- ☐ `CURRENT_MONTH` — текущий номер месяца с предваряющим нулем для одной цифры;
- ☐ `CURRENT_MONTH_NAME` — название текущего месяца (например, "Сентябрь");
- ☐ `CURRENT_MONTH_NAME_SHORT` — сокращенное название текущего месяца (например, "Сен");
- ☐ `CURRENT_DATE` — день в месяце;
- ☐ `CURRENT_DAY_NAME` — полное название дня недели (например, "среда");
- ☐ `CURRENT_DAY_NAME_SHORT` — сокращенное название дня недели (например, "ср");
- ☐ `CURRENT_HOUR` — текущий час в 24-часовом формате;

- ❑ `CURRENT_MINUTE` — минуты с предваряющим нулем для одной цифры (например, 08);
- ❑ `CURRENT_SECOND` — секунды с предваряющим нулем для одной цифры (например, 02);
- ❑ `CURRENT_SECONDS_UNIX` — число секунд, прошедшее с 1 января 1970 года;
- ❑ `RANDOM` — псевдослучайное 6-значное десятичное число;
- ❑ `RANDOM_HEX` — псевдослучайное 6-значное шестнадцатеричное число;
- ❑ `UUID` — универсальный уникальный идентификатор версии 4 (псевдослучайный);
- ❑ `BLOCK_COMMENT_START` — открывающая комбинация символов для вставки многострочного комментария. Значение зависит от текущего формата файла. В HTML-файле вставляется комбинация `<!--`;
- ❑ `BLOCK_COMMENT_END` — закрывающая комбинация символов для вставки многострочного комментария. Значение зависит от текущего формата файла. В HTML-файле вставляется комбинация `-->`;
- ❑ `LINE_COMMENT` — комбинация символов для вставки однострочного комментария. Значение зависит от текущего формата файла. В JavaScript-файле вставляется комбинация `//`.

Пример вставки текущей даты:

```
"Текущая дата": {
  "prefix": "curdate",
  "body": "$CURRENT_DATE.$CURRENT_MONTH.$CURRENT_YEAR",
  "description": "Текущая дата"
}
```

Примерный результат:

23.04.2021

Имя переменной также можно обернуть в фигурные скобки:

```
"body": "${CURRENT_DATE}.${CURRENT_MONTH}.${CURRENT_YEAR}"
```

После имени переменной допускается указание двоеточия, после которого задается значение по умолчанию, используемое, если переменная не содержит значения:

```
"body": "${TM_SELECTED_TEXT:Значение по умолчанию}"
```

Если нужно изменить значение переменной перед вставкой значения, то после переменной указывается следующая конструкция:

```
</Регулярное выражение>/<Замена>/[<Модификатор>]
```

В первом параметре указывается шаблон регулярного выражения языка JavaScript для поиска совпадений. Во втором параметре задается строка для замены найденных совпадений. В третьем параметре задаются модификаторы регулярного выражения. Например, чтобы выполнить замену всех вхождений, нужно указать флаг `g`.

Чтобы не учитывался регистр символов, следует добавить флаг `i`. Заменим все обратные слешы, двоеточие и точку в пути символом подчеркивания:

```
"body": "${TM_FILEPATH/[\\\:\\.]/_}/g"
```

Примерный результат:

```
C__book_app_js
```

Изначально значение переменной выглядело так:

```
C:\book\app.js
```

Обратите внимание на количество защитных слешей в регулярном выражении. Обратный слеш в строке является специальным, поэтому его нужно экранировать дополнительным обратным слешем. Внутри регулярного выражения обратный слеш опять является специальным, поэтому количество слешей удваивается. В итоге получаем четыре слеша для вставки в шаблон всего одного. Символ "точка" также является специальным в шаблоне и соответствует любому символу (возможно, кроме символа перевода строки). Однако если точку указать внутри квадратных скобок, то она теряет свое специальное предназначение, поэтому в нашем примере она не экранирована.

В строке для замены можно использовать обратные ссылки на группы внутри регулярного выражения, заключенные в круглые скобки. Порядковый номер группы в шаблоне указывается после символа `$` (например, `$1`). Нумерация начинается с 1. После номера можно указать двоеточие и способ изменения регистра символов:

☐ `/uppercase` — переводит все буквы в верхний регистр:

```
"body": "${TM_FILEPATH/([a-z]+)/${1:/uppercase}/g}"
```

Результат:

```
C:\BOOK\APP.JS
```

☐ `/downcase` — переводит все буквы в нижний регистр:

```
"body": "${TM_FILEPATH/([A-Z]+)/${1:/downcase}/g}"
```

Результат:

```
c:\book\app.js
```

☐ `/capitalize` — переводит первую букву каждого слова в верхний регистр:

```
"body": "${TM_FILEPATH/([a-z]+)/${1:/capitalize}/gi}"
```

Результат:

```
C:\Book\App.Js
```

Пользовательские фрагменты кода можно назначить для всех типов файлов. При этом фрагменты кода могут работать как на глобальном уровне (могут быть вставлены в любой файл, хранящийся в любом каталоге), так и локально, только на уровне каталога, открытого в данный момент.

Для создания "глобальных" фрагментов кода в меню **Файл** выбираем пункт **Настройки | Пользовательские фрагменты кода**. Отобразится панель **Палитра команд** (см. рис. 1.29). Из списка выбираем пункт **Новый файл с глобальным фрагментом кода**. В текстовое поле вводим имя файла без расширения (например, `mysnippets`) и нажимаем клавишу `<Enter>`. В результате в каталоге `C:\Users\<Имя пользователя>\AppData\Roaming\Code\User\snippets` будет создан файл `mysnippets.code-snippets`.

ОБРАТИТЕ ВНИМАНИЕ

Расширение файла `code-snippets`, а не `json`.

Удаляем все содержимое файла и вставляем следующий код:

```
{
  "Текущая дата": {
    "prefix": "curdate",
    "body": "$CURRENT_DATE.$CURRENT_MONTH.$CURRENT_YEAR",
    "description": "Текущая дата"
  }
}
```

Теперь комбинация `curdate` будет работать во всех файлах. Если требуется ограничить перечень типов файлов, то необходимо добавить свойство `scope` и в нем перечислить нужные типы через запятую. Ограничимся HTML- и JavaScript-файлами:

```
{
  "Текущая дата": {
    "scope": "html,javascript",
    "prefix": "curdate",
    "body": "$CURRENT_DATE.$CURRENT_MONTH.$CURRENT_YEAR",
    "description": "Текущая дата"
  }
}
```

"Локальные" фрагменты кода создаются так же, как и "глобальные", за тем исключением, что в **Палитре команд** нужно выбрать пункт **Создать файл фрагментов для '<имя открытого каталога>'**. Файлы с такими фрагментами кода сохранятся в каталоге `.vscode`, вложенном в открытый каталог.

1.12. Работа с курсорами и выделениями

Редактор VS Code позволяет создать сразу несколько текстовых курсоров. Для этого необходимо сделать несколько щелчков в нужных местах кода, удерживая нажатой клавишу `<Alt>`. Теперь при вводе с клавиатуры текст будет вставляться сразу во всех местах, помеченных курсорами.

Точно так же можно выделить сразу несколько фрагментов с помощью мыши, удерживая при этом нажатой клавишу `<Alt>`. При вводе с клавиатуры текст будет заменяться сразу во всех местах, помеченных выделениями.

Чтобы установить курсоры в конец каждой выделенной строки, следует нажать комбинацию клавиш <Shift>+<Ctrl>+<I>.

Для выделения текущей строки нужно нажать комбинацию клавиш <Ctrl>+<L>. Чтобы выделить весь текст, следует нажать комбинацию клавиш <Ctrl>+<A>. Чтобы расширить или сжать текущее выделение, удерживая нажатой клавишу <Shift>, нажмите клавишу с нужной стрелкой.

Если при наличии выделенного фрагмента нажать комбинацию клавиш <Shift>+<Ctrl>+<L>, то будут выделены все вхождения текущего выделения. А если нажать комбинацию <Ctrl>+<D>, будет выделен только следующий фрагмент, совпадающий с текущим выделением. Например, таким способом можно выделить открывающий и закрывающий теги, а затем переименовать их. Каждое последующее нажатие комбинации клавиш <Ctrl>+<D> будет добавлять к выделению следующий фрагмент.

Для выделения вертикального блока текста нужно при перемещении мыши с нажатой левой кнопкой удерживать нажатой комбинацию клавиш <Shift>+<Alt>.

Чтобы быстро перейти в начало файла, нажмите комбинацию клавиш <Ctrl>+<Home>, для быстрого перехода в конец файла — <Ctrl>+<End>.

Редактор VS Code также поддерживает стандартные комбинации клавиш:

- <Ctrl>+<X> — **Вырезать**;
- <Ctrl>+<C> — **Копировать**;
- <Ctrl>+<V> — **Вставить**;
- <Ctrl>+<Z> — **Отменить**;
- <Ctrl>+<Y> — **Повторить**.

Все эти команды доступны также в меню **Правка**, а первые три — еще и в контекстном меню.

ПРИМЕЧАНИЕ

Чтобы увидеть все комбинации клавиш и изменить их, в меню **Файл** выберите пункт **Настройки | Сочетания клавиш** или нажмите комбинацию клавиш <Ctrl>+<K>, а затем комбинацию <Ctrl>+<S>. Обратите также внимание на пункты в меню **Выделение** (см. рис. 1.13).

1.13. Дублирование, перемещение и удаление строк

Для вставки копии текущей строки под ней следует нажать комбинацию клавиш <Shift>+<Alt>+<стрелка вниз>. Такой способ очень удобен при добавлении новых пунктов списка. Комбинация клавиш <Shift>+<Alt>+<стрелка вверх> позволяет вставить копию выше текущей строки.

Чтобы продублировать выделенный фрагмент кода, следует в меню **Выделение** выбрать пункт **Дублировать выделенное**.

Чтобы переместить текущую строку ниже или выше, следует нажать комбинации клавиш `<Alt>+<стрелка вниз>` и `<Alt>+<стрелка вверх>` соответственно.

Быстро удалить строку, на которой стоит текстовый курсор, можно нажатием комбинации клавиш `<Ctrl>+<Shift>+<K>`.

1.14. Сворачивание и разворачивание блоков кода

Если навести указатель мыши немного левее самого первого символа строки кода, но правее нумерации строк, то отобразятся значки в виде стрелок вниз для открывающих фрагментов блоков. Сейчас в HTML-файле у нас три блока со значками: открывающий тег `<html>`, открывающий тег `<head>` и открывающий тег `<body>`. Если в секцию тела (`<body>`) были добавлены другие блочные теги с вложениями элементов, то для них также отобразятся значки. Если щелкнуть левой кнопкой мыши на значке, блок свернется. При этом значок примет вид стрелки, показывающей вправо. Повторный щелчок на значке вновь развернет блок.

Аналогичные действия можно выполнить с помощью комбинаций клавиш `<Shift>+<Ctrl>+<[>` (свернуть блок) и `<Shift>+<Ctrl>+<]>` (развернуть блок).

Чтобы свернуть сразу все блоки, следует нажать комбинацию клавиш `<Ctrl>+<K>`, а затем комбинацию клавиш `<Ctrl>+<0>` (цифра "ноль" на основной клавиатуре). Для разворачивания сразу всех свернутых блоков нужно нажать комбинацию клавиш `<Ctrl>+<K>`, а затем комбинацию клавиш `<Ctrl>+<J>`.

1.15. Изменение регистра символов

Чтобы изменить регистр символов, выделяем нужный фрагмент, а затем открываем **Палитру команд**, нажав комбинацию клавиш `<Shift>+<Ctrl>+<P>`. В текстовое поле вводим "transform" (или "регистр") и выбираем нужный пункт из списка (рис. 1.30).

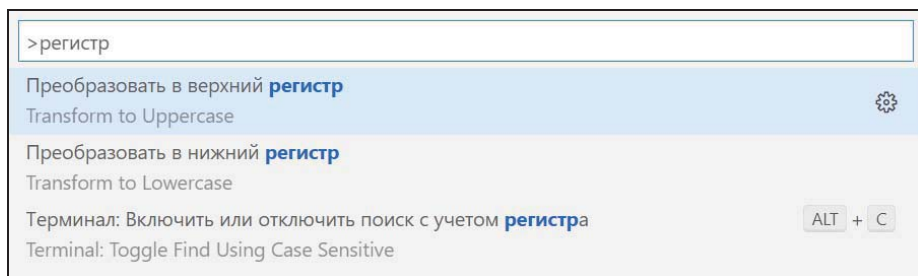


Рис. 1.30. Палитра команд: изменение регистра символов

1.16. Изменение масштаба

Для изменения масштаба всех элементов редактора, включая меню, используются следующие комбинации клавиш:

- ☐ `<Ctrl>+<=>` — увеличивает масштаб. Также можно в меню **Вид** выбрать пункт **Внешний вид | Увеличить**;
- ☐ `<Ctrl>+<->` — уменьшает масштаб (в меню **Вид** пункт **Внешний вид | Уменьшить**);
- ☐ `<Ctrl>+<0>` (цифра "ноль" на цифровой клавиатуре) — сбрасывает все изменения масштаба к первоначальным значениям (в меню **Вид** пункт **Внешний вид | Сбросить масштаб**).

Для того чтобы изменить масштаб только содержимого вкладок, удерживая нажатой клавишу `<Ctrl>`, поворачиваем в нужную сторону колесико мыши. Данная функция будет доступна, если вы установили флажок **Editor: Mouse Wheel Zoom** в настройках редактора (см. *разд. 1.2*).

1.17. Полноэкранный режим

Редактор VS Code предоставляет три полноэкранных режима:

- ☐ развернутый режим — для перехода в этот режим нажимаем кнопку **Развернуть** в заголовке окна. В этом режиме доступны все компоненты редактора, а также панель задач Windows. Для выхода из режима нажимаем кнопку **Восстановить** в заголовке окна;
- ☐ полноэкранный режим — для перехода в этот режим в меню **Вид** выбираем пункт **Внешний вид | Полный экран** или нажимаем клавишу `<F11>`. В этом режиме доступны только компоненты редактора, которые занимают всю доступную область экрана, строка меню и панель задач Windows не отображаются. Для выхода из режима нажимаем клавишу `<F11>`;
- ☐ режим Zen — для перехода в этот режим в меню **Вид** выбираем пункт **Внешний вид | Режим Zen** или нажимаем комбинацию клавиш `<Ctrl>+<K>`, а затем клавишу `<Z>`. В этом режиме доступно только содержимое текущей вкладки, выровненное по центру внутри области с боковыми границами, которые можно перемещать, изменяя ширину области просмотра. Окно занимает всю доступную область экрана. Строка меню, панели редактора и панель задач Windows не отображаются. Для выхода из режима нажимаем комбинацию клавиш `<Ctrl>+<K>`, а затем клавишу `<Z>` или дважды нажимаем клавишу `<Esc>`.

1.18. Форматирование кода

Редактор VS Code поддерживает автоматическое форматирование кода. Чтобы отформатировать содержимое активной вкладки, в контекстном меню выбираем пункт **Форматировать документ** или нажимаем комбинацию клавиш `<Shift>+`

+<Alt>+<F>. Можно также выделить фрагмент и в контекстном меню выбрать пункт **Форматировать выделенный фрагмент** или нажать комбинацию клавиш <Ctrl>+<K>, а затем <Ctrl>+<F>.

В настройках редактора можно найти много пунктов, задающих отдельные настройки форматирования. Если установлен флажок **Editor: Format On Paste**, будет произведено автоматическое форматирование кода при вставке из буфера обмена. Если установлен флажок **Editor: Format On Save**, будет автоматически форматироваться весь документ при сохранении.

Чтобы увеличить отступ слева у отдельной строки, следует поставить в ее начало текстовый курсор и нажать клавишу <Tab>. Нажатие комбинации клавиш <Shift>+<Tab> уменьшает отступ слева.

Аналогично можно увеличивать и уменьшать отступ слева у целых фрагментов кода. Для этого достаточно выделить нужный фрагмент и нажать соответственно клавишу <Tab> или комбинацию клавиш <Shift>+<Tab>.

Рекомендуем дополнительно установить расширения Prettier — Code formatter и Beautify. Они содержат большое количество настроек, и, вполне возможно, результат их работы понравится вам больше, чем "художества" VS Code. После установки в контекстном меню появятся пункты, позволяющие выбрать инструмент для форматирования с помощью панели **Палитра команд**. В настройках редактора можно также указать способ форматирования по умолчанию.

ПРИМЕЧАНИЕ

Описание установки и настройки Prettier приведено в разд. 13.9.

1.19. Полезные расширения

Редактор VS Code позволяет устанавливать сторонние расширения, делающие работу с кодом значительно удобнее. Некоторые из них мы рассмотрели в предыдущих разделах. Рекомендуем также обратить внимание на следующие расширения:

- ☐ AutoFileName — выводит всплывающие подсказки со списками, перечисляющими файлы, которые входят в состав проекта. Значительно упрощает запись значений в параметры href и src HTML-тегов;
- ☐ Bracket Pair Colorizer — раскрашивает скобки разными цветами. Это очень удобно при работе с кодом на языке JavaScript и форматом JSON;
- ☐ HTML CSS Support — "подсказывает" имена стилевых классов при их вводе в параметрах class в HTML-коде. Очень пригодится в случае больших таблиц стилей, например при использовании библиотеки Bootstrap. Просматривает все внешние таблицы стилей в каталоге проекта;
- ☐ All Autocomplete — выводит всплывающие подсказки со всеми словами из всех открытых документов. Если открыть на отдельной вкладке таблицу стилей библиотеки Bootstrap, то выводимая подсказка будет содержать все стилевые классы этой библиотеки;

- ❑ CSS Peek — позволяет быстро переходить к определениям стилей и даже редактировать их внутри HTML-документа (изменения при этом сохраняются в CSS-файле);
- ❑ SCSS IntelliSense — отображает подсказки при работе с SCSS-файлами, например "подсказывает" имена переменных, объявленных в другом модуле, при вводе символа `$`;
- ❑ Live Sass Compiler — выполняет автоматическое преобразование файлов CSS-препроцессора Sass в CSS-файлы при сохранении;
- ❑ Bootstrap v4 Snippets — содержит готовые фрагменты кода из библиотеки Bootstrap 4. Список с названиями фрагментов отображается при вводе префикса `b-`;
- ❑ Bootstrap 5 Snippets — содержит готовые фрагменты кода из библиотеки Bootstrap 5. Список с названиями поддерживаемых фрагментов приведен в описании расширения;
- ❑ jQuery Code Snippets — содержит готовые фрагменты кода из библиотеки jQuery. Список с названиями фрагментов отображается при вводе префикса `j`;
- ❑ Polacode — создает красивые скриншоты с фрагментами кода. Такие изображения не стыдно опубликовать на форуме или в блоге.

Следует учитывать, что некоторые расширения замедляют работу редактора, поэтому устанавливать слишком много расширений не рекомендуется. Если расширение в данный момент не нужно, его можно временно отключить на странице расширения, а впоследствии включить вновь.

В этой главе мы рассмотрели основные возможности редактора VS Code. В последующих главах мы будем рассматривать дополнительные инструменты редактора применительно к языку JavaScript и платформе Node.js.



ГЛАВА 2

Node.js: первые шаги

Прежде чем мы начнем рассматривать синтаксис языка, необходимо сделать два замечания. Во-первых, не забывайте, что книги по программированию нужно не только читать, но и выполнять все приведенные в них примеры, а также экспериментировать, что-либо в этих примерах изменяя. Во-вторых, помните, что прочитать книгу один раз недостаточно — ее вы должны выучить наизусть! Это основы языка!

2.1. Установка Node.js

В этой книге мы рассмотрим основы языка JavaScript на примере создания консольных программ, работающих под управлением платформы Node.js. И сначала установим эту платформу.

Node.js поставляется в двух редакциях: LTS (с длительным периодом поддержки, наиболее стабильная, предназначенная для эксплуатации) и Current (текущая, включающая все последние нововведения, но не столь стабильная). Мы будем использовать редакцию LTS.

Переходим на сайт <https://nodejs.org/> и щелкаем на гиперссылке **Other Downloads** под большой зеленой кнопкой с надписью **<номер версии> LTS**. После перехода на страницу загрузок скачиваем программу установки Windows Installer 64-bit (для 64-разрядной редакции Windows) или Windows Installer 32-bit (для 32-разрядной редакции). В нашем случае файл установщика будет иметь имя `node-v16.0.0-x64.msi` или `node-v16.0.0-x86.msi` соответственно.

Запускаем программу установки и на первом шаге нажимаем кнопку **Next** (рис. 2.1). На втором шаге соглашаемся с лицензионным соглашением и нажимаем кнопку **Next** (рис. 2.2). На следующем шаге нужно выбрать каталог для установки Node.js (рис. 2.3). Можно оставить путь по умолчанию, но мы установим Node.js в каталог `C:\nodejs\node16`, чтобы в пути не было пробелов. Указываем каталог и нажимаем кнопку **Next**. На следующем шаге можно выбрать устанавливаемые компоненты (рис. 2.4). Оставляем настройки по умолчанию и нажимаем кнопку **Next**. На следующем шаге (рис. 2.5) также ничего не меняем и нажимаем кнопку **Next**.

Далее нажимаем кнопку **Install** (рис. 2.6) для запуска процесса установки. Для установки требуются права администратора, поэтому в следующем окне нужно разрешить установку. Нажимаем кнопку **Finish** (рис. 2.7) для завершения работы программы установки.

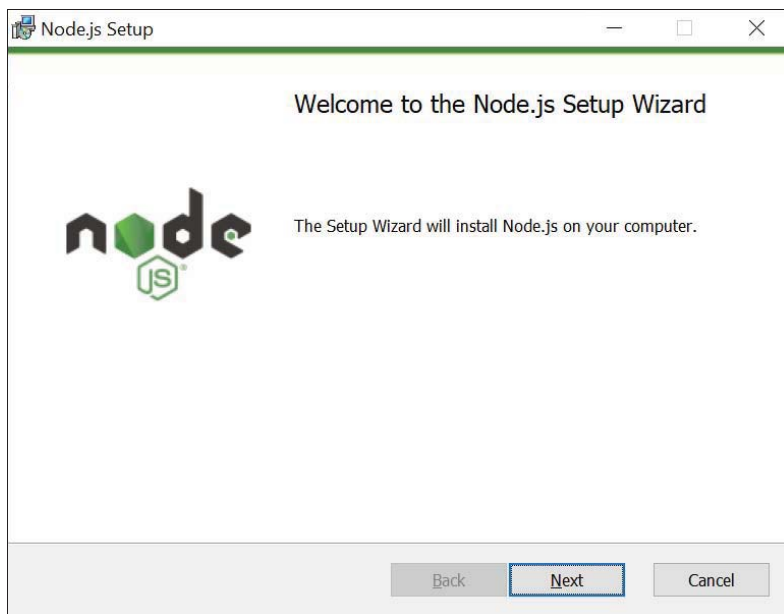


Рис. 2.1. Установка Node.js: шаг 1

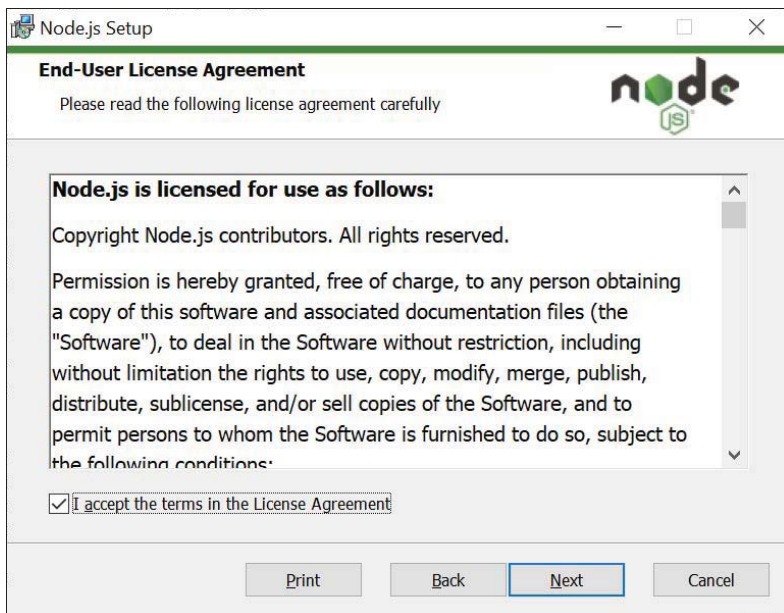


Рис. 2.2. Установка Node.js: шаг 2

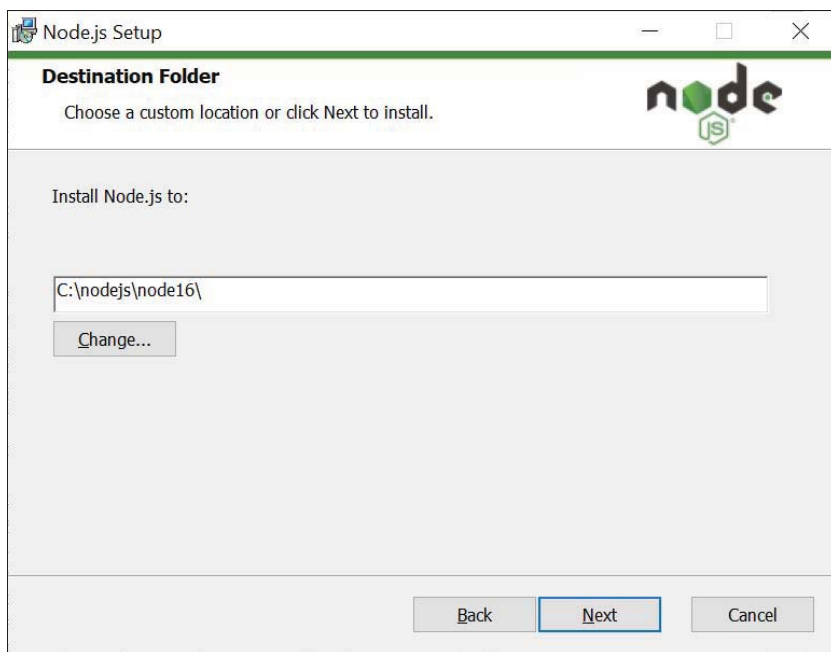


Рис. 2.3. Установка Node.js: шаг 3

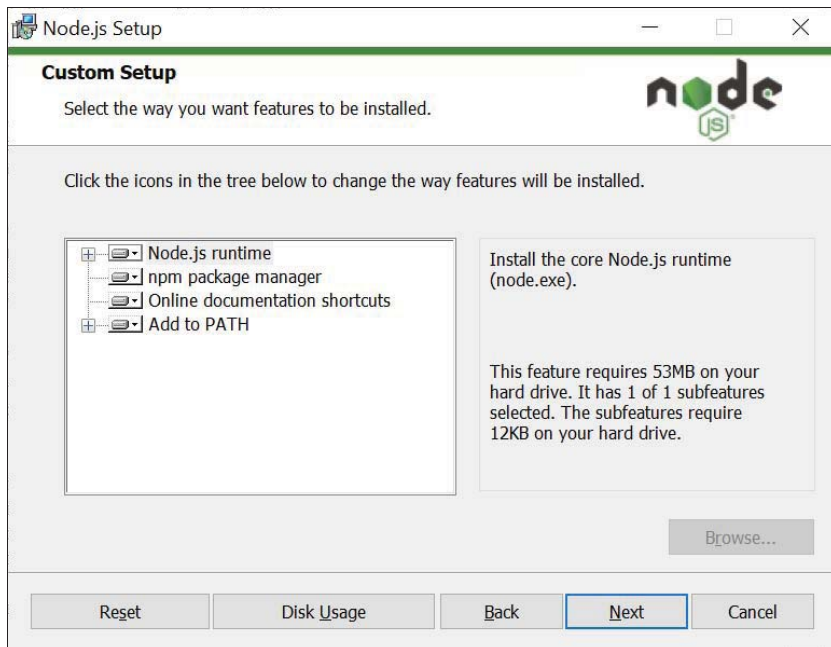


Рис. 2.4. Установка Node.js: шаг 4

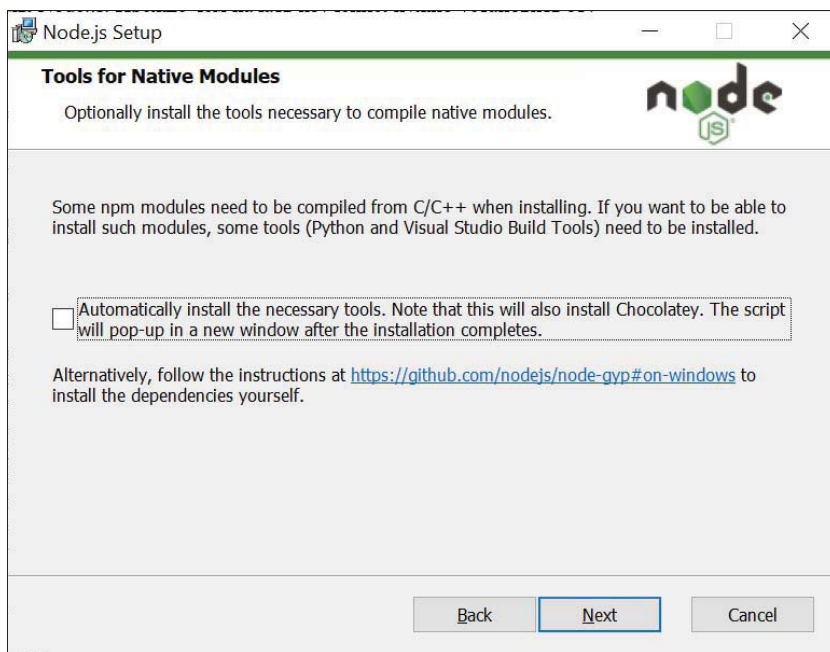


Рис. 2.5. Установка Node.js: шаг 5

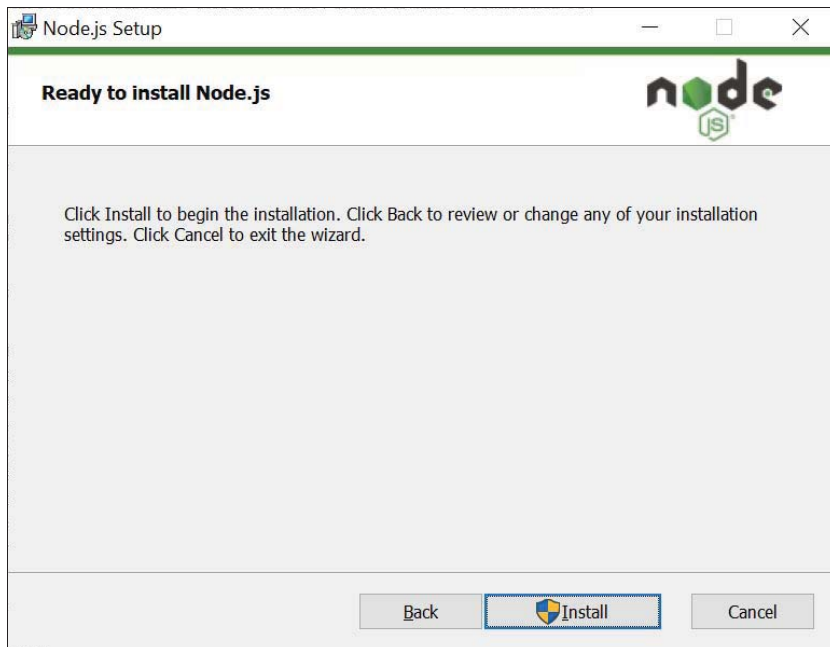


Рис. 2.6. Установка Node.js: шаг 6

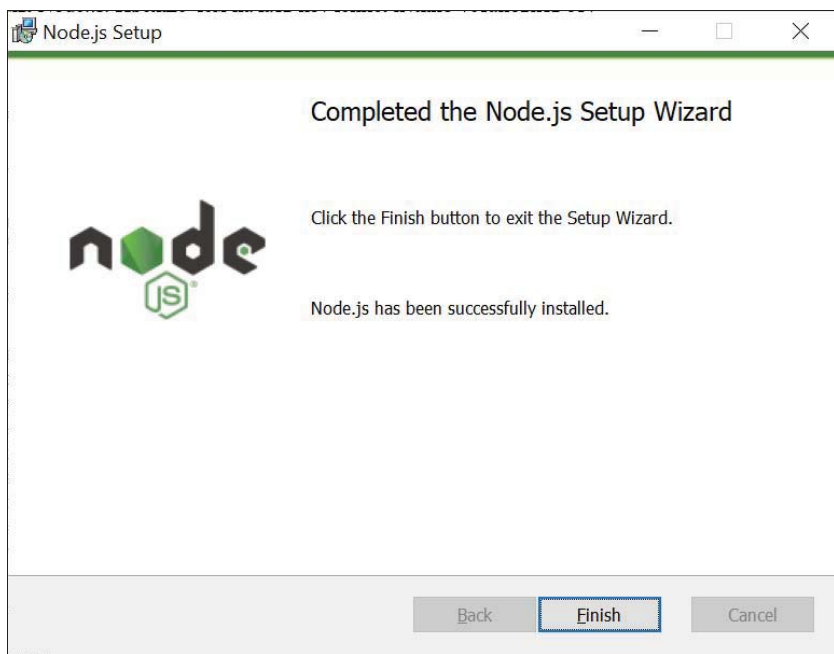


Рис. 2.7. Установка Node.js: шаг 7

2.2. Работа с командной строкой

Для дальнейшей работы нам понадобится командная строка. Запустить ее можно несколькими способами:

- ☐ выполнив в меню **Пуск** поиск по фразе "командная строка";
- ☐ нажав комбинацию клавиш <Windows>+<R>, введя в открывшемся окне "cmd" и нажав кнопку **ОК**;
- ☐ запустив файл cmd.exe в каталоге C:\Windows\System32;
- ☐ в Проводнике щелкнув правой кнопкой мыши на свободном месте списка файлов, удерживая при этом нажатой клавишу <Shift>, и из контекстного меню выбрав пункт **Открыть окно команд**;
- ☐ в Проводнике в адресной строке введя "cmd" и нажав клавишу <Enter>.

В некоторых случаях для выполнения различных команд могут потребоваться права администратора. Чтобы запустить командную строку с правами администратора, нужно щелкнуть на значке этого приложения правой кнопкой мыши и выбрать в появившемся контекстном меню пункт **Запуск от имени администратора**.

Запомните способы запуска командной строки наизусть. В дальнейшем мы просто будем говорить "запустите командную строку" без уточнения, как это сделать.

Платформа Node.js устанавливается в каталоге, указанном на шаге 3 (см. рис. 2.3). Кроме того, если вы ничего не меняли на шаге 4 (см. рис. 2.4), путь к Node.js до-

бавляется в переменную окружения `PATH`. Поэтому в командной строке можно просто указать название программы без явного добавления пути к ней. Давайте проверим работоспособность Node.js. Запускаем командную строку и выполняем команду `node --version`:

```
C:\book>node --version
v16.0.0
```

Можно также воспользоваться командой `node -v`:

```
C:\book\p1>node -v
v16.0.0
```

Фрагмент `C:\book>` перед командой означает приглашение для ввода команды с указанием текущего рабочего каталога. Текст после команды является результатом ее выполнения. В нашем примере мы получили версию Node.js.

Если вы получили следующий результат:

```
C:\book>node --version
"node" не является внутренней или внешней
командой, исполняемой программой или пакетным файлом.
```

значит, программа Node.js не найдена. Вполне возможно, что путь не был добавлен в переменную окружения `PATH`. Выведите значение этой переменной с помощью команды:

```
echo %Path%
```

Если путь не прописан, его нужно добавить вручную. Чтобы изменить системную переменную `PATH` в Windows, переходим в **Параметры | Панель управления | Система и безопасность | Система | Дополнительные параметры системы**. В результате откроется окно **Свойства системы**. На вкладке **Дополнительно** нажимаем кнопку **Переменные среды**. В открывшемся окне в списке **Системные переменные** выделяем строку с переменной `Path` и нажимаем кнопку **Изменить**.

ВНИМАНИЕ!

Случайно не удалите существующее значение переменной `PATH`, иначе другие приложения перестанут запускаться.

Значение системной переменной можно временно изменить для текущего сеанса, т. е. до момента закрытия приложения Командная строка. Для этого перед работой с Node.js нужно выполнить следующую команду:

```
set Path=<Путь до Node.js>;%Path%
```

Таким же способом можно запускать другие версии Node.js без переустановки. Путь добавляется в самое начало системной переменной `PATH`, поэтому будет использоваться именно этот путь. На компьютере одного из авторов уставлено несколько версий Node.js, и он может запустить, например, Node.js версии 14:

```
C:\book>set Path=C:\nodejs\node14;%Path%

C:\book>node --version
v14.16.1
```

Если в пути есть пробелы, то в командной строке такой путь нужно обязательно указывать в кавычках, иначе он будет обрезан до первого пробела. Поэтому, чтобы избежать проблем, на третьем шаге при установке лучше указывать путь без пробелов. Кроме того, имена каталогов и файлов в пути не должны содержать русских букв. Допустимы только латинские буквы, цифры, дефис и подчеркивание.

Советуем также убедиться, что имя пользователя компьютера содержит только латинские буквы и цифры. Никаких русских букв и пробелов, т. к. многие программы, включая Node.js, сохраняют различные настройки и временные файлы в каталоге `C:\Users\<Имя пользователя>`. Если имя пользователя содержит русские буквы, то они могут быть искажены до неузнаваемости из-за неправильного преобразования кодировок и программа не сможет сохранить настройки. Помните, что в разных кодировках русские буквы могут иметь разный код. Разработчики программ в основном работают с английским языком и ничего не знают о проблемах с кодировками, т. к. во всех однобайтовых кодировках и в UTF-8 коды латинских букв одинаковы. Так что, если хотите без проблем заниматься программированием, то от использования русских букв в имени пользователя лучше отказаться.

2.3. Первая программа на JavaScript

В процессе изучения основ языка JavaScript мы будем создавать консольные приложения. *Консольное приложение* — это программа, отображающая текстовую информацию и позволяющая вводить символы с клавиатуры. Консольное приложение позволит не отвлекаться на освоение среды разработки, а полностью сосредоточить внимание на изучении синтаксиса языка.

При изучении языков программирования принято начинать с программы, выводящей надпись "Привет, мир!" Не будем нарушать традицию и продемонстрируем, как это выглядит на JavaScript (листинг 2.1).

Листинг 2.1. Первая программа

```
// Выводим надпись с помощью метода log()
console.log('Привет, мир!');
```

Сохраняем код из листинга 2.1 в кодировке UTF-8 без BOM в файле `C:\book\app.js`, удалив предыдущее содержимое этого файла. Запускаем командную строку и выполняем следующую команду:

```
C:\Users\Unicross>node C:\book\app.js
Привет, мир!
```

В результате мы получили приветствие, причем на русском языке без проблем с кодировками. Преобразование кодировок при выводе данных производится автоматически. Вы же помните, что наш файл сохранен в кодировке UTF-8? А консоль использует по умолчанию кодировку Windows-866. Давайте в этом убедимся:

```
C:\Users\Unicross>chcp
```

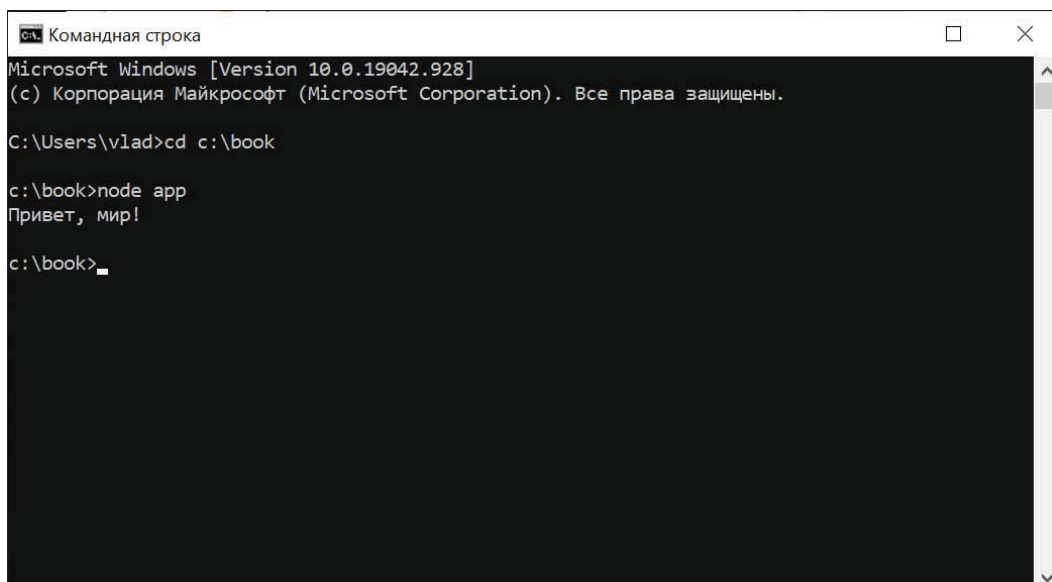
Текущая кодовая страница: 866

Чтобы не вводить каждый раз полный путь к программе, следует сделать каталог C:\book текущим с помощью следующей команды:

```
C:\Users\Unicross>cd C:\book
```

Теперь достаточно указать только имя файла с программой, можно без расширения (рис. 2.8):

```
C:\book>node app  
Привет, мир!
```



```
Командная строка
Microsoft Windows [Version 10.0.19042.928]
(c) Корпорация Майкрософт (Microsoft Corporation). Все права защищены.

C:\Users\vlad>cd c:\book

c:\book>node app
Привет, мир!

c:\book>
```

Рис. 2.8. Запуск программы в приложении Командная строка

НА ЗАМЕТКУ

Все выполненные команды записываются в список истории. Перемещаться внутри этого списка позволяют клавиши со стрелками вверх и вниз. Например, чтобы выполнить последнюю команду, достаточно нажать клавишу со стрелкой вверх, а затем клавишу <Enter>. Чтобы очистить консоль, нужно ввести команду `cls`. Список истории при этом не очищается.

2.4. Вкладка *Терминал* в редакторе VS Code

Запускать программы на выполнение также можно на вкладке **Терминал**, расположенной на нижней панели (см. *разд. 1.4.4*) редактора VS Code. Для отображения вкладки **Терминал** в меню **Вид** (см. рис. 1.14) выбираем пункт **Терминал** или нажимаем комбинацию клавиш <Ctrl>+<обратный апостроф>.

Если в редакторе открыт каталог C:\book, то в терминале будет следующее приглашение для ввода команд:

```
PS C:\book>
```

По умолчанию во вкладке **Терминал** запускается приложение PowerShell, пришедшее на смену устаревшей командной строке. "Визитной карточкой" этого приложения являются буквы "PS", находящиеся в начале приглашения к вводу команды, перед путем к текущему каталогу. PowerShell поддерживает все команды, исполняемые командной строкой, и вдобавок предоставляет богатую дополнительную функциональность (рассказ о которой выходит за рамки этой книги).

Запустить программу на выполнение можно так же, как и в приложении Командная строка (рис. 2.9):

```
PS C:\book> node app
```

```
Привет, мир!
```

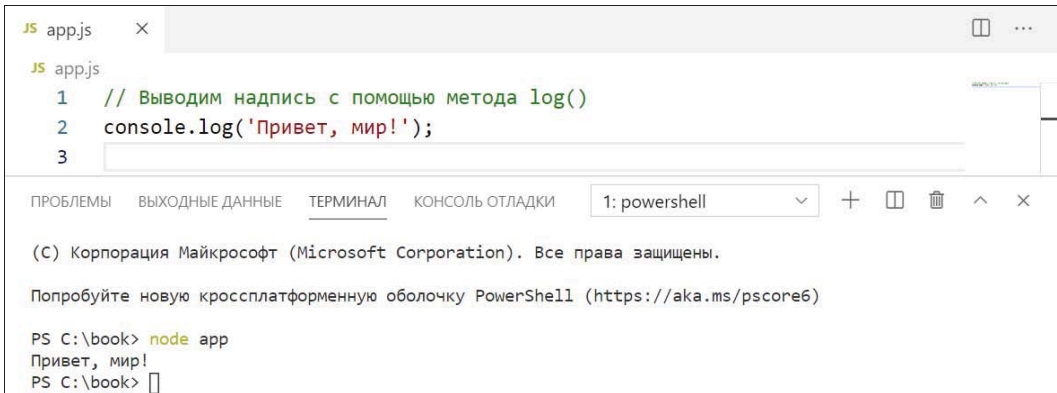


Рис. 2.9. Запуск программы на вкладке **Терминал** редактора VS Code

Для вставки команды из буфера обмена достаточно щелкнуть правой кнопкой мыши. Если существует выделенный фрагмент, то щелчок правой кнопкой мыши скопирует фрагмент в буфер обмена. Эти действия задаются следующей настройкой:

```
"terminal.integrated.rightClickBehavior": "copyPaste"
```

В ряде случаев на вкладке **Терминал** мы будем запускать команду, которая будет выполняться постоянно, блокируя ввод других команд. Чтобы иметь возможность выполнять при этом другие команды, нужно создать новый терминал. Для этого в меню **Терминал** выбираем пункт **Создать терминал** или нажимаем комбинацию клавиш <Shift>+<Ctrl>+<обратный апостроф>. Для быстрого создания терминала достаточно щелкнуть на значке с изображением символа "плюс" в заголовке нижней панели (рис. 2.10). Этими способами можно создать сколько угодно терминалов.

Для переключения между терминалами следует выбрать нужный пункт из списка, расположенного в заголовке нижней панели (см. рис. 2.10). Для закрытия терминала щелкаем на значке в виде корзины, расположенном там же.

НА ЗАМЕТКУ

При необходимости на вкладке **Терминал** можно запустить и "старую" командную строку Windows, и любую другую аналогичную программу (например, bash.exe, расположенную в каталоге bin системы управления версиями Git).

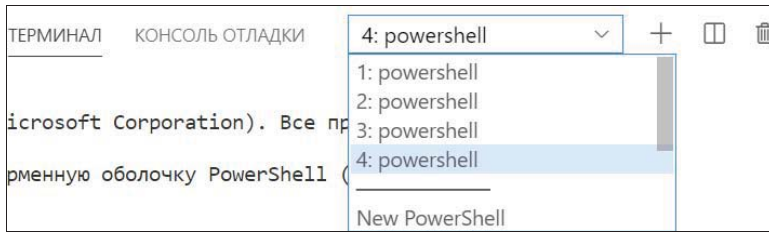


Рис. 2.10. Переключение между терминалами

2.5. NPM (Node Package Manager)

В комплект поставки Node.js входит программа *NPM (Node Package Manager)*, менеджер пакетов Node.js), позволяющая устанавливать различные библиотеки, утилиты и управлять разрабатываемым проектом. Давайте проверим ее работоспособность, отобразив версию:

```
C:\book>npm --version
7.10.0
```

Можно также воспользоваться командой `npm -v`:

```
C:\book>npm --v
7.10.0
```

Для вывода полной информации выполняем следующую команду:

```
C:\book>npm version
{
  npm: '7.10.0',
  node: '16.0.0',
  v8: '9.0.257.17-node.10',
  uv: '1.41.0',
  zlib: '1.2.11',
  brotli: '1.0.9',
  ares: '1.17.1',
  modules: '93',
  nghttp2: '1.42.0',
  napi: '8',
  llhttp: '6.0.0',
  openssl: '1.1.1k+quic',
  cldr: '39.0',
  icu: '69.1',
  tz: '2021a',
  unicode: '13.0',
  ngtcp2: '0.1.0-DEV',
  nghttp3: '0.1.0-DEV'
}
```

Если вы успешно получили версию NPM, то можно продолжить изучение материала.

2.5.1. Создание пакета и добавление файла package.json

Создадим полноценный пакет Node.js под именем p1 (сокращение от package1, чтобы команды были короче). В Проводнике Windows в каталоге C:\book создаем вложенный каталог p1, а затем открываем его в редакторе VS Code. Все дальнейшие действия по созданию файлов и каталогов выполняем на панели **Проводник** (см. *разд. 1.5*). В каталоге C:\book\p1 создаем файл index.js и вставляем в него код из листинга 2.1.

Чтобы превратить каталог C:\book\p1 в полноценный пакет Node.js, нужно создать в этом каталоге файл package.json. Для этого открываем командную строку и переходим в каталог пакета:

```
C:\Users\Unicross>cd C:\book\p1
```

```
C:\book\p1>
```

ОБРАТИТЕ ВНИМАНИЕ

Имя каталога должно быть в составе приглашения для ввода команды. Если путь будет иным, файл будет создан в другом каталоге.

Для создания файла package.json выполняем команду `npm init -y`:

```
c:\book\p1>npm init -y
```

```
Wrote to c:\book\p1\package.json:
```

```
{
  "name": "p1",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

После выполнения команды было выведено содержимое созданного файла.

В данной команде мы задали флаг `-y`. Если его не указать, NPM будет запрашивать необходимые данные в командной строке. Можно ввести данные, а можно на каждый запрос нажимать клавишу `<Enter>` — в этом случае будут использованы значения по умолчанию. В конце диалога нужно подтвердить создание файла. Пример:

```
C:\book\p1>npm init
```

```
This utility will walk you through creating a package.json file.
```

```
It only covers the most common items, and tries to guess sensible defaults.
```

See ``npm help init`` for definitive documentation on these fields and exactly what they do.

Use ``npm install <pkg>`` afterwards to install a package and save it as a dependency in the `package.json` file.

Press `^C` at any time to quit.

package name: (p1)

version: (1.0.0)

description:

git repository:

author:

license: (ISC)

About to write to `C:\book\p1\package.json`:

```
{
  "name": "p1",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": ""
}
```

Is this OK? (yes)

Гораздо удобнее отредактировать созданный файл в текстовом редакторе, чем что-то вводить в командной строке. Поэтому в команде мы указали флаг `-y`. Вместо него можно указать флаг `--yes`:

```
C:\book\p1>npm init --yes
```

При использовании формата JSON следует учитывать следующие моменты:

- ☐ данные хранятся в кодировке UTF-8;
- ☐ все содержимое файла `package.json` заключается в фигурные скобки, внутри которых через запятую указываются пары:

`<Свойство>: <Значение>`

- ☐ имена свойств и строки заключаются в двойные кавычки. Если в составе строки присутствует кавычка, ее нужно экранировать обратным слешем. Пример:

```
"test": "echo \"Error: no test specified\" && exit 1"
```

- ☐ значением свойства могут быть только число, строка, массив (его элементы указываются внутри квадратных скобок) или объект (его свойства указываются внутри фигурных скобок);

- ❑ отредактировать файл можно любым текстовым редактором. Чтобы открыть файл на вкладке редактора VS Code, достаточно щелкнуть на имени файла на панели **Проводник**.

Рассмотрим основные свойства внутри файла `package.json`:

- ❑ `name` — имя пакета;
- ❑ `version` — версия пакета. Нумерация версий должна соответствовать следующим правилам:
 - 1.0.0 — первая версия пакета;
 - 1.0.1 — изменение третьего числа означает исправление ошибок с обратной совместимостью;
 - 1.1.0 — изменение второго числа означает добавление новой функциональности с обратной совместимостью;
 - 2.0.0 — изменение первого числа означает изменение кода, возможно, без обратной совместимости.

При изменении файлов проекта следует сменить версию пакета;

- ❑ `main` — имя главного исполняемого файла пакета. По умолчанию файл имеет имя `index.js`. Если хотите использовать файл с другим именем, соответственно отредактируйте значение этого свойства;
- ❑ `scripts` — объект с запускаемыми скриптами. По умолчанию при создании файла `package.json` добавляется скрипт `test`, который выводит сообщение в консоль и завершает свое выполнение ошибкой. Подробно создание и запуск скриптов мы рассмотрим далее в этой главе;
- ❑ `keywords` — массив с ключевыми словами, с помощью которых можно найти пакет в *репозитории* (интернет-хранилище библиотек и дополнительных утилит Node.js);
- ❑ `author` — информация об авторе;
- ❑ `license` — лицензия на использование пакета. Если хотите сделать пакет доступным всем желающим, выберите свободную лицензию MIT;
- ❑ `description` — описание пакета.

Существуют и другие свойства, которые мы будем рассматривать по мере изучения материала.

Для запуска программы, находящейся в составе пакета, из командной строки сначала делаем каталог `C:\book\p1` текущим, а затем выполняем команду:

```
C:\book\p1>node index  
Привет, мир!
```

Имя запускаемого JavaScript-файла можно указывать без расширения.

Все команды выполняем в корневом каталоге пакета. При использовании терминала в VS Code каталог пакета делается текущим по умолчанию, если он открыт в редакторе.

2.5.2. Установка пакетов

Написанный пакет можно добавить в репозиторий, и любой желающий сможет загрузить и установить его с помощью программы NPM. Установка выполняется командой `npm install` (или `npm i`) тремя основными способами:

- глобально — указанием в команде флага `-g` или `--global`. Установленный глобально пакет будет доступен для всех проектов. Установим глобально препроцессор `sass`, реализованный в виде одноименного пакета:

```
C:\book\p1>npm install -g sass
```

Файлы установленного пакета `sass` будут записаны в каталогах `C:\Users\<Имя пользователя>\AppData\Roaming\npm` и `C:\Users\<Имя пользователя>\AppData\Roaming\npm\node_modules\sass`. Первый каталог добавлен в переменную окружения `PATH`, поэтому мы можем в командной строке указывать только имя программы без пути к ней. Выведем версию `sass`:

```
C:\book\p1>sass --version
1.32.11 compiled with dart2js 2.12.4
```

При инсталляции какой-либо программы или библиотеки также будут установлены все *зависимости* — библиотеки, используемые ею в работе;

- на уровне пакета как зависимость, необходимую на этапах разработки и выполнения, — указанием в команде флага `--save` или `-S` или введя команду вообще без флагов. Установим библиотеки `Bootstrap` и `jQuery`:

```
C:\book\p1>npm install bootstrap --save
C:\book\p1>npm install jquery
```

Библиотеки будут записаны в каталог `C:\book\p1\node_modules`. При этом в файл `package.json` автоматически добавится свойство `dependencies` с перечнем установленных пакетов и их версий:

```
"dependencies": {
  "bootstrap": "^4.6.0",
  "jquery": "^3.6.0"
}
```

Впоследствии, при установке текущего пакета из репозитория, все записанные в файле `package.json` зависимости будут установлены автоматически;

- на уровне пакета как зависимость, необходимую лишь на этапе разработки, — указанием в команде флага `--save-dev` или `-D`. Установим пакет `Webpack`:

```
npm install webpack --save-dev
```

В каталоге `C:\book\p1\node_modules` будет создан каталог `webpack` с файлами установленной программы. А в файл `package.json` будет добавлено свойство `devDependencies` с перечнем установленных пакетов и их версий:

```
"devDependencies": {
  "webpack": "^5.36.0"
}
```

При установке текущего пакета такие зависимости не будут установлены автоматически, поскольку для выполнения пакета они не нужны.

За один раз можно установить сразу несколько пакетов, перечислив их через пробел. Установим пакеты `webpack-cli`, `css-loader` и `sass-loader`, применяемые при разработке:

```
C:\book\p1>npm install webpack-cli css-loader sass-loader --save-dev
```

Значение свойства `devDependencies` стало таким:

```
"devDependencies": {  
  "css-loader": "^5.2.4",  
  "sass-loader": "^11.0.1",  
  "webpack": "^5.36.0",  
  "webpack-cli": "^4.6.0"  
}
```

В каталоге пакета `C:\book\p1` автоматически создается файл `package-lock.json`, в котором описываются все установленные пакеты, включая зависимости. Не следует изменять этот файл вручную. Вывести список всех установленных пакетов позволяет команда:

```
C:\book\p1>npm list
```

Чтобы посмотреть только зависимости текущего проекта, можно воспользоваться следующей командой:

```
C:\book\p1>npm list --depth 0
```

Флаг `--depth` просто сворачивает все вложенные зависимости. Для вывода списка всех пакетов, установленных глобально, следует указать флаг `-g`:

```
C:\book\p1>npm list -g
```

Получение списка глобальных пакетов только верхнего уровня:

```
C:\book\p1>npm list -g --depth 0
```

Обратите внимание на количество каталогов, вложенных в `C:\book\p1\node_modules`. Мы установили всего четыре пакета, а они "подтянули" еще почти 300 пакетов-зависимостей. Иногда общий объем установленных зависимостей достигает гигабайта! Не зря каталог `node_modules` называют "черной дырой"...

Вы наверняка обратили внимание на то, что перед версией пакета добавлен символ `^`. Он означает, что может быть установлен пакет указанной или более новой версии при условии, что изменяются только вторые или третьи числа в версии, т. е. при сохранении совместимости версий. Вместо символа `^` можно указать символ `~`, который позволяет изменять только третье число в версии.

Для установки конкретной версии ее нужно указать после символа `@`:

```
npm install <Пакет>@<Версия>
```

Установим библиотеку `Popper.js`, используемую `Bootstrap` для позиционирования всплывающих элементов (например, подсказок и меню). Как нам узнать требуемую

версию? Правильно! Открыть файл `package.json`, расположенный в каталоге `C:\book\p1\node_modules\bootstrap`, и посмотреть раздел с зависимостями:

```
"popper.js": "^1.16.1"
```

Итак, нам нужна версия 1.16.1. Устанавливаем ее:

```
npm install popper.js@1.16.1
```

Проверяем установку и видим, что в свойстве `dependencies` добавлена строка с библиотекой `Popper.js` той версии, которая нам и нужна:

```
"dependencies": {  
  "bootstrap": "^4.5.2",  
  "jquery": "^3.5.1",  
  "popper.js": "^1.16.1"  
}
```

Благодаря свойствам `dependencies` (этап работы) и `devDependencies` (этап разработки) все пользователи нашего пакета знают о необходимых зависимостях и их версиях. Можно полностью удалить каталог `node_modules`, и команда `npm install` (без указания имени пакета) целиком восстановит его содержимое на основе сведений из файла `package.json`.

Вместо версии можно указать теги, например `latest` для установки последней версии или `next` для установки тестовых "ночных" сборок пакетов. Например, установим "ночную" сборку компилятора языка TypeScript на глобальном уровне:

```
C:\book\p1>npm install -g typescript@next
```

Выведем версию компилятора в консоли:

```
C:\book\p1>tsc -v  
Version 4.3.0-dev.20210420
```

Получить список всех доступных для загрузки версий пакета `typescript` позволяет следующая команда:

```
C:\book\p1>npm view typescript versions
```

2.5.3. Обновление и удаление пакетов

Выполнить проверку на наличие устаревших версий пакетов позволяет команда `npm outdated`. Если устаревших пакетов нет, то никакого вывода не будет, в противном случае получим список. Проверим пакеты глобального уровня:

```
C:\book\p1>npm -g outdated
```

Package	Current	Wanted	Latest	Location
sass	1.32.11	1.32.12	1.32.12	global
typescript	4.3.0-dev.20210420	4.3.0-dev.20210420	4.2.4	global

В списке выводятся установленные версии, найденные версии и версии с тегом `latest`. Как можно заметить из результата, вышла новая версия пакета `sass`. Чтобы обновить пакет, следует выполнить команду:

```
npm update [-g] [<Пакет>]
```

Если имя пакета не указано, будут обновлены все устаревшие пакеты. Пример обновления только пакета sass, установленного на глобальном уровне:

```
C:\book\pl>npm update -g sass
```

Выведем устаревшие пакеты еще раз:

```
C:\book\pl>npm -g outdated
Package           Current           Wanted   Latest   Location
typescript  4.3.0-dev.20210420  4.3.0-dev.20210420  4.2.4   global
```

Как видно из результата пакета sass в списке устаревших пакетов больше нет. Выведем версию sass в консоли:

```
C:\book\pl>sass --version
1.32.12 compiled with dart2js 2.12.4
```

Для удаления пакета предназначена команда:

```
npm uninstall [-g] <Пакет> [--save|--save-dev|--save-optional|--no-save]
```

Удалим пакет typescript из глобального уровня:

```
npm uninstall -g typescript
```

Проверим удаление пакета путем вывода всех установленных пакетов на глобальном уровне:

```
C:\book\pl>npm list -g --depth 0
C:\Users\Unicross\AppData\Roaming\npm
`-- sass@1.32.12
```

2.5.4. Добавление и запуск скриптов

Программа NPM, помимо управления пакетами, позволяет запускать скрипты, записанные в свойстве `scripts` файла `package.json`. Для этого используется следующая команда:

```
npm run <Имя скрипта> [-- <Параметры>]
```

Сейчас свойство `scripts` в файле `package.json` имеет такое значение:

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1"
}
```

Изменим скрипт `test` так, чтобы он запускал на выполнение файл `index.js`:

```
"scripts": {
  "test": "node index.js"
}
```

Запустим скрипт из командной строки:

```
C:\book\pl>npm run test
> pl@1.0.0 test C:\book\pl
> node index.js
```

Привет, мир!

При возникновении ошибки ее описание выводится в консоли. Кроме того, в каталоге `C:\Users\<Имя пользователя>\AppData\Roaming\npm-cache\_logs` создается файл с полным описанием ошибки. Не забывайте периодически чистить этот каталог.

В предыдущих разделах мы установили пакет `Webpack`. Если мы сейчас попробуем вывести его версию в командной строке, у нас ничего не получится:

```
C:\book\p1>webpack --version
"webpack" не является внутренней или внешней
командой, исполняемой программой или пакетным файлом.
```

При глобальной установке пакета получить версию можно, т.к. каталог `C:\Users\<Имя пользователя>\AppData\Roaming\npm` добавлен в переменную окружения `PATH`. Выведем версию пакета `sass`:

```
C:\book\p1>sass --version
1.32.12 compiled with dart2js 2.12.4
```

При запуске скрипта с помощью `NPM` в переменную окружения `PATH` добавляется путь к каталогу `node_modules\.bin`, в котором расположены запускаемые скрипты для пакета. Поэтому с помощью скрипта можно получить версию `Webpack`. Добавим скрипт `webpack-version` в файл `package.json`:

```
"scripts": {
  "test": "node index.js",
  "webpack-version": "webpack --version"
}
```

Теперь получим версию `Webpack`:

```
C:\book\p1>npm run webpack-version
```

```
> pl@1.0.0 webpack-version
> webpack --version
```

```
webpack 5.36.0
webpack-cli 4.6.0
```

Напишем скрипты для преобразования `SCSS`-файлов в `CSS`-файлы с помощью `sass`. Внутри каталога `C:\book\p1` создадим каталоги `scss` и `dist`. В каталоге `C:\book\p1\dist` создадим вложенный каталог `css`, в который будут сохраняться создаваемые `CSS`-файлы. В каталоге `C:\book\p1\scss` создадим файл `main.scss` (листинг 2.2), содержащий код таблицы стилей на языке `SCSS`. Файл должен быть сохранен в кодировке `UTF-8` без `BOM`.

Листинг 2.2. Содержимое файла `main.scss`

```
$black: #000000;
$white: #ffffff;

body {
  color: $white;
  background-color: $black;
}
```

Теперь добавим в файл `package.json` следующие скрипты:

```
"scripts": {
  "test": "node index.js",
  "webpack-version": "webpack --version",
  "sass-dev": "sass scss:dist/css",
  "sass-watch": "sass --watch scss:dist/css"
}
```

Скрипт `sass-dev` преобразует все SCSS-файлы в каталоге `C:\book\p1\scss` в одноименные CSS-файлы и сохраняет их в каталоге `C:\book\p1\dist\css`:

```
C:\book\p1>npm run sass-dev
```

```
> p1@1.0.0 sass-dev C:\book\p1
> sass scss:dist/css
```

После выполнения скрипта `sass-dev` в каталоге `C:\book\p1\dist\css` будут созданы файлы `main.css` и `main.css.map`. Содержимое файла `main.css`:

```
body {
  color: #ffffff;
  background-color: #000000;
}

/*# sourceMappingURL=main.css.map */
```

Скрипт `sass-watch` запускает отслеживание изменений SCSS-файлов внутри каталога `C:\book\p1\scss`:

```
C:\book\p1>npm run sass-watch
```

```
> p1@1.0.0 sass-watch C:\book\p1
> sass --watch scss:dist/css
```

```
Compiled scss\main.scss to dist\css\main.css.
Sass is watching for changes. Press Ctrl-C to stop.
```

После запуска скрипта программа `sass` будет работать постоянно, блокируя ввод других команд. Чтобы получить возможность вводить другие команды, нужно запустить дополнительное окно командной строки. Сразу после сохранения SCSS-файла автоматически будет запущен процесс преобразования и в командной строке появится следующее сообщение, выделенное зеленым цветом:

```
Compiled scss\main.scss to dist\css\main.css.
```

Если в файле содержится ошибка, будет выведено сообщение о проблеме:

```
Error: Undefined variable.
```

```
|   background-color: $black2;
|                       ^^^^^^^
```

```
scss\main.scss 6:22  root stylesheet
```

Чтобы завершить программу, нужно нажать комбинацию клавиш <Ctrl>+<C> или закрыть консоль. В первом случае программа запросит подтверждение действия путем ввода буквы "y" и нажатия клавиши <Enter>:

Завершить выполнение пакетного файла [Y(да)/N(нет)]? y

C:\book\p1>

Запускать скрипты можно не только в командной строке, но и в редакторе VS Code. Для этого используются три основных способа:

1. В файле `package.json` навести указатель мыши на имя скрипта. Отобразится всплывающее окно с двумя пунктами: **Запуск сценария** и **Отладка сценария**. Нужно щелкнуть гиперссылку **Запуск сценария** в этом окне. Результат выполнения скрипта отобразится на вкладке **Терминал**.

Также можно щелкнуть правой кнопкой мыши на скрипте и выбрать в появившемся контекстном меню пункт **Запуск сценария**;

2. На панели **Проводник** в секции **Сценарии NPM** (рис. 2.11) навести указатель мыши на имя скрипта. Справа появятся два значка: **Запуск** и **Отладка**. Следует щелкнуть на значке **Запуск**. Результат выполнения скрипта отобразится на вкладке **Терминал**.

Также можно щелкнуть правой кнопкой мыши на скрипте и выбрать в появившемся контекстном меню пункт **Запуск**.

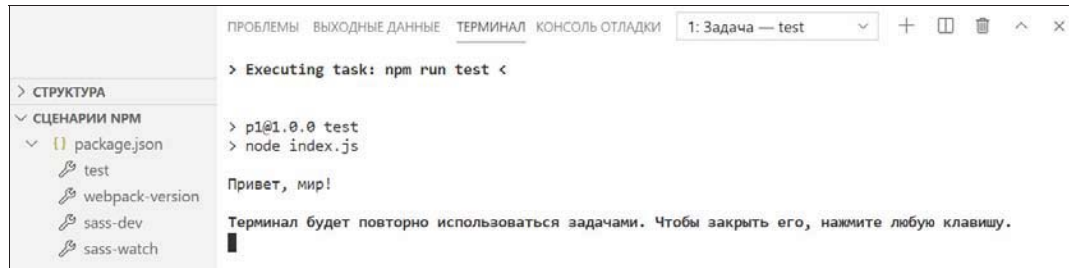


Рис. 2.11. Секция **Сценарии NPM** на панели **Проводник** и запуск скрипта в терминале

Если на панели **Проводник** отсутствует раздел **Сценарии NPM**, щелкните на кнопке со значком многоточия, которая находится правее надписи "Проводник" вверху панели. После чего выберите пункт **Сценарии NPM** в появившемся на экране небольшом меню.

3. В меню **Терминал** выбрать пункт **Запуск задачи**, щелкнуть на ссылке **Все задачи**, найти нужный скрипт в появившемся списке и щелкнуть на нем.

Результат выполнения скрипта отобразится в новом терминале, открытом на вкладке **Терминал** в нижней панели. Закрыть этот терминал можно нажатием любой клавиши.

Еще можно создать командный файл с расширением `bat` и запускать нужный скрипт (или скрипты) из него. После инструкции запуска программы вставляем команду

pause, которая выведет строку "Для продолжения нажмите любую клавишу" и будет ожидать нажатия. Давайте в каталоге C:\book\p1 создадим файл script.bat со следующим кодом:

```
@echo off
title Запуск программы index.js
echo Результат:
@echo.
node index.js
@echo.
@echo.
@echo.
pause
```

ОБРАТИТЕ ВНИМАНИЕ

Кодировка файла script.bat должна быть cp866, а не UTF-8.

После запуска скрипта откроется окно, изображенное на рис. 2.12, с результатом выполнения программы index.js. Для закрытия окна достаточно нажать любую клавишу.

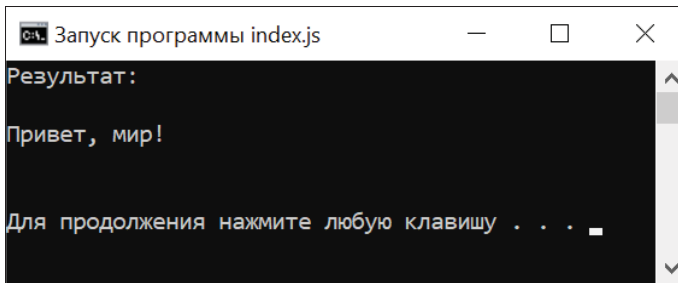


Рис. 2.12. Результат запуска программы с помощью файла script.bat

Существуют и другие способы запуска программы. Например, с помощью веб-браузера. Однако тут возможны проблемы, т. к. мы будем изучать все последние нововведения в языке JavaScript, которые веб-браузеры еще не поддерживают. Поэтому запуск программ в веб-браузере мы рассмотрим отдельно.

2.6. Структура программы

Программа состоит из инструкций, расположенных в текстовом файле, сохраненном в кодировке UTF-8 без BOM (без метки порядка байтов). Каждая инструкция располагается на отдельной строке. В листинге 2.1 инструкций две:

- первая строка содержит однострочный комментарий, который начинается с символов //, заканчивается в конце строки и описывает назначение следующей инструкции:

```
// Выводим надпись с помощью метода log()
```

- вторая строка содержит инструкцию, выводящую строку "Привет, мир!" в консоль:

```
console.log('Привет, мир!');
```

Строка заключается в одинарные или двойные кавычки. Одинарную кавычку набрать с клавиатуры проще, поэтому мы рекомендуем именно этот способ.

Строка передается внутри круглых скобок методу `log()` встроенного объекта `console`. Объект — это одна из программных подсистем, из которых состоит Node.js, а метод — это один из инструментов, предоставляемых этой подсистемой. Так, объект `console` — это консоль Node.js, в которой пользователь вводит какие-либо данные и читает результат, выводимый программой, а метод `log()` — это инструмент для вывода в консоли заданных строк.

Имена объекта и метода разделяются точкой. Значение, передаваемое методу (параметр), записывается в круглых скобках, следующих непосредственно за именем метода. Если метод принимает несколько параметров, они указываются через запятую:

```
console.log('Строка 1', 'Строка 2');
```

Любая инструкция в JavaScript должна заканчиваться точкой с запятой.

Как правило, каждая инструкция записывается в отдельной строке — такой программный код лучше читается. Но допускается записывать в одной строке сразу несколько инструкций, если они короткие:

```
console.log('Строка 1'); console.log('Строка 2')
```

Можно записать инструкцию, состоящую из одной лишь точки с запятой:

```
;
```

Такая инструкция называется "пустой", ничего не делает и применяется в очень редких случаях (например, если тело цикла не содержит инструкций).

При запуске программы все инструкции выполняются слева направо и сверху вниз интерпретатором языка JavaScript, в качестве которого может выступать Node.js или веб-браузер. С помощью операторов ветвления и циклов можно изменить порядок выполнения инструкций. Эти операторы будут рассмотрены в следующих главах.

2.7. Комментарии в JavaScript

Комментарии предназначены для вставки пояснений в программный код. Интерпретатор полностью их игнорирует. Внутри комментария может располагаться любой текст (включая инструкции, которые выполнять не следует).

Язык JavaScript поддерживает два типа комментариев: однострочный и многострочный. *Однострочный комментарий* начинается с символов `//` и заканчивается в конце строки. Вставлять однострочный комментарий можно как в начале строки, так и после инструкции. Если символы `//` разместить перед инструкцией, то она не

будет выполнена. Если символы `//` расположены внутри кавычек, то они не являются признаком начала комментария. Примеры однострочных комментариев:

```
// Это комментарий
console.log('Привет, мир!');    // Это комментарий
// console.log('Привет, мир!'); // Инструкция выполнена не будет
console.log('// Это НЕ комментарий!!!');
```

Многострочный комментарий начинается с символов `/*` и заканчивается символами `*/`. Комментарий может быть расположен как на одной строке, так и на нескольких. Кроме того, многострочный комментарий можно размещать внутри выражения, хотя это и нежелательно. Следует иметь в виду, что многострочные комментарии не могут быть вложенными, поэтому при комментировании больших блоков следует проверять, что в них не встречается закрывающая комментарий комбинация символов `*/`. Тем не менее однострочный комментарий может быть расположен внутри многострочного комментария. Примеры многострочных комментариев:

```
/*
 * Многострочный комментарий
 */
console.log('Привет, мир!');    /* Это комментарий */
/* console.log('Привет, мир!'); // Инструкция выполнена не будет */
let x;
x = 10 /* Комментарий */ + 50 /* внутри выражения */;
console.log(x); /* 60 */
console.log('/* Это НЕ комментарий!!! */');
```

Редактор VS Code позволяет быстро создавать комментарии. Чтобы превратить строку в однострочный комментарий, нужно поставить на эту строку текстовый курсор и нажать комбинацию клавиш `<Ctrl>+</>`. Если предварительно выделить несколько строк, все они превратятся в однострочные комментарии. Если все выделенные строки были закомментированы ранее, то комбинация клавиш `<Ctrl>+</>` раскомментирует их. Можно также в меню **Правка** выбрать пункт **Переключить комментарий строки**.

Для вставки многострочного комментария необходимо выделить строки и нажать комбинацию клавиш `<Shift>+<Alt>+<A>`. Повторное нажатие этой комбинации клавиш раскомментирует строки. Можно также в меню **Правка** выбрать пункт **Переключить комментарий блока**.

У начинающих программистов может возникнуть вопрос, зачем может потребоваться комментировать инструкции? Дело в том, что часто в логике программы возникают проблемы по вине программиста (например, программа выдает неверный результат). Чтобы найти ошибку, приходится отключать часть кода с помощью комментариев, вставлять инструкции вывода промежуточных результатов и анализировать их.

2.8. Интерактивный режим

Инструкции JavaScript можно выполнять в командной строке и сразу видеть результат, запустив интерактивный режим. Сделать это можно, набрав в командной строке команду `node`:

```
C:\book\pl>node
Welcome to Node.js v16.0.0.
Type ".help" for more information.
>
```

После запуска выводится приглашение для ввода инструкции `>`, и программа ожидает нашего ввода. Вводим инструкцию или выражение и нажимаем клавишу `<Enter>`. На следующей строке сразу выводится результат выполнения инструкции, и программа снова ожидает наших действий:

```
> console.log('Привет, мир!');
Привет, мир!
undefined
>
```

Во второй строке вывелось сообщение, а в третьей — результат, возвращаемый методом `log()`, — значение `undefined`, информирующее об отсутствии возвращаемого значения.

В предыдущем примере мы выводили строку с помощью метода `log()`. В интерактивном режиме это делать не обязательно. Мы можем просто ввести строку и нажать клавишу `<Enter>` для получения результата. Пример:

```
> 'Привет, мир!'
'Привет, мир!'
>
```

ОБРАТИТЕ ВНИМАНИЕ

Строка вывелась в одинарных кавычках, тогда как при использовании метода `log()` кавычки не выводятся.

Учитывая возможность получить результат сразу после ввода команды, интерактивный режим пригоден для изучения команд, а также в качестве многофункционального калькулятора:

```
> 10 + 12 * 2
34
>
```

Однако следует учитывать особенности работы с вещественными числами. В этом примере ожидаемым был бы результат `0.0`, но мы получаем совсем другое:

```
> 0.3 - 0.1 - 0.2
-2.7755575615628914e-17
>
```

Результат вычисления последнего выражения сохраняется в переменной с именем `_` (одно подчеркивание). Это позволяет выполнять дальнейшие расчеты без ввода предыдущего результата:

```
> 25 * 3           // Умножение
75
> _ + 25           // Сложение. Эквивалентно 75 + 25
100
> _ / 20           // Деление. Эквивалентно 100 / 20
5
```

Для выхода из интерактивного режима дважды нажимаем комбинацию клавиш `<Ctrl>+<C>` или выполняем команду `.exit`:

```
(To exit, press ^C again or ^D or type .exit)
>
```

```
C:\book\p1>
```

2.9. Вывод результатов работы программы

Вывести результаты работы программы в консоли позволяет метод `log()` объекта `console`. Метод имеет следующие форматы:

```
console.log()
console.log(<Значение 1>[, . . . , <Значение N>])
console.log(<Строка формата>, <Значение 1>[, . . . , <Значение N>])
```

Первый формат выводит в стандартный поток `stdout` (т. е. в консоль) символ перевода строки:

```
console.log('Строка 1');
console.log();
console.log('Строка 3');
```

Результат:

```
Строка 1
```

```
Строка 3
```

Второй формат выводит в стандартный поток `stdout` указанные значения и в конце добавляет перевод строки:

```
console.log('Строка 1');
console.log('Строка 2');
```

Результат:

```
Строка 1
```

```
Строка 2
```

Если указано несколько значений, между ними выводятся пробелы:

```
console.log('Строка', 1, 2, 3); // Строка 1 2 3
```

В третьем формате в первом параметре указывается строка формата. В ней можно указать обычные символы и спецификаторы формата, начинающиеся с символа %. Вместо спецификаторов формата будут подставлены заданные значения. Пример:

```
console.log('Count %d', 10); // Count 10
console.log('%s %d', 'Count', 10); // Count 10
```

В первой инструкции внутри строки формата используется спецификатор %d, предназначенный для вывода числа. Вместо этого спецификатора подставляется число 10, переданное во втором параметре. Во второй инструкции строка формата содержит сразу два спецификатора %s и %d. Спецификатор %s предназначен для вывода строки, а спецификатор %d — для вывода числа. Вместо спецификатора %s будет подставлена строка 'Count', а вместо спецификатора %d — число 10.

Чтобы вывести символ %, который в строке формата является специальным, его нужно удвоить:

```
console.log('%d%%', 10); // 10%
```

ПРИМЕЧАНИЕ

Полный список спецификаторов приведен в документации.

Вместо метода `log()` можно использовать полностью аналогичный ему метод `info()`:

```
console.info('Строка 1'); // Строка 1
console.info('Строка 2'); // Строка 2
console.info('Count', 10); // Count 10
console.info('Count %d', 10); // Count 10
```

Для вывода массивов и объектов в табличной форме предназначен метод `table()` объекта `console`:

```
let arr = [1, 2, 3];
console.table(arr);
```

Результат:

(index)	Values
0	1
1	2
2	3

Пример вывода структуры объекта:

```
let obj = {a: 5, b: 2};
console.table(obj);
```

Результат:

(index)	Values
a	5
b	2

Вывести сообщение об ошибке позволяет метод `error()` объекта `console`. Метод имеет следующие форматы:

```
console.error(<Значение 1>[, . . . , <Значение N>])
console.error(<Строка формата>, <Значение 1>[, . . . , <Значение N>])
```

Первый формат выводит указанные значения в стандартный поток `stderr` и в конце добавляет перевод строки:

```
console.error('Сообщение об ошибке 1');
console.error('Сообщение об ошибке 2');
```

Результат в консоли:

```
Сообщение об ошибке 1
Сообщение об ошибке 2
```

Если указано несколько значений, между ними выводятся пробелы:

```
console.error('Код ошибки:', 10); // Код ошибки: 10
```

Во втором формате в первом параметре указывается строка формата. В ней могут присутствовать обычные символы и спецификаторы формата, начинающиеся с символа `%`. Вместо спецификаторов формата при выводе будут подставлены заданные значения. Пример:

```
console.error('Код ошибки: %d', 10); // Код ошибки: 10
```

Вместо метода `error()` можно использовать полностью аналогичный ему метод `warn()`:

```
console.warn('Сообщение об ошибке 1'); // Сообщение об ошибке 1
console.warn('Сообщение об ошибке 2'); // Сообщение об ошибке 2
console.warn('Код ошибки:', 10);       // Код ошибки: 10
console.warn('Код ошибки: %d', 10);    // Код ошибки: 10
```

VS Code предоставляет следующие комбинации для быстрого ввода:

```
log<Tab>  => console.log();
error<Tab>=> console.error();
warn<Tab> => console.warn();
```

Все рассмотренные методы после вывода данных вставляют символ перевода строки. Если он не нужен, можно воспользоваться методом `write(<Строка>)` потоков `stdout` (стандартный поток вывода обычных сообщений) или `stderr` (стандартный поток вывода сообщений об ошибках). Эти потоки реализованы в объекте `process`. Выведем строки с помощью метода `write()`:

```
process.stdout.write('Строка');  
process.stdout.write('1\n');  
process.stdout.write('Строка2');
```

Результат:

```
Строка1  
Строка2
```

Метод `clear()` объекта `console` очищает консоль:

```
console.clear();
```

2.10. Получение данных из командной строки

Передать данные в программу можно, указав их при запуске этой программы в командной строке после имени запускаемого файла. Получить эти данные в программе позволяет свойство `argv` объекта `process` (свойство — это ячейка памяти, принадлежащая объекту, хранящая какое-либо значение и имеющая уникальное имя).

Свойство `argv` содержит массив строк. *Массив* — это нумерованный набор элементов, расположенных в памяти последовательно друг за другом. Позиция элемента внутри массива задается с помощью *индекса*. Нумерация элементов массива начинается с 0, а не с 1. Получить число элементов (*размер*) массива позволяет его свойство `length`. Выведем содержимое и размер массива в консоли:

```
console.log(process.argv);  
console.log(process.argv.length);
```

После запуска программы получим примерно следующий результат:

```
C:\book\p1>node index.js  
[ 'C:\nodejs\node16\node.exe', 'C:\book\p1\index.js' ]  
2
```

По умолчанию массив из свойства `argv` содержит два элемента:

- первый элемент (имеет индекс 0) — путь к программе `node.exe`. Получить этот путь позволяют также свойства `argv0` и `execPath` объекта `process`:

```
console.log(process.argv0);  
console.log(process.argv[0]);  
console.log(process.execPath);
```

Значение свойства `argv0` зависит от способа запуска в командной строке (содержит исходное значение `argv[0]`), тогда как свойство `execPath` всегда содержит абсолютный путь к программе `node.exe`:

```
C:\book\p1>node index.js  
node  
C:\nodejs\node16\node.exe  
C:\nodejs\node16\node.exe
```



```
C:\book\p1>node.exe index.js
node.exe
C:\nodejs\node16\node.exe
C:\nodejs\node16\node.exe
```

```
C:\book\p1>C:\nodejs\node16\node.exe index.js
C:\nodejs\node16\node.exe
C:\nodejs\node16\node.exe
C:\nodejs\node16\node.exe
```

□ второй элемент (имеет индекс 1) — путь к запускаемому файлу:

```
console.log(process.argv[1]); // C:\book\p1\index.js
```

Теперь рассмотрим получение данных из командной строки (листинг 2.3).

Листинг 2.3. Получение данных из командной строки

```
const count = process.argv.length;
console.log('Количество элементов:', count);
for (let i = 0; i < count; i++) {
  console.log(process.argv[i]);
}
```

В листинге 2.3 использован цикл `for`. Инструкции, расположенные в этом цикле внутри блока (ограниченного фигурными скобками), выполняются несколько раз. Число повторений зависит от выражений внутри круглых скобок. Этих выражений три, и разделены они точками с запятой. Первое выражение объявляет переменную `i` (ее можно рассматривать как аналог свойства) и присваивает ей значение 0. Второе выражение является условием продолжения повторений. Пока значение переменной `i` меньше значения `count` (количества параметров, переданных программе), инструкции внутри блока будут повторяться. Это условие проверяется на каждой итерации цикла. Третье выражение на каждой итерации цикла прибавляет 1 к текущему значению переменной `i`. Доступ к элементу массива осуществляется с помощью текущего индекса, указанного внутри квадратных скобок. Благодаря циклу мы можем получить любое количество данных, переданных в командной строке.

Для запуска программы вводим команду:

```
C:\book\p1>node index.js -param1 -param2
```

В этой команде мы передаем программе некоторые данные (`-param1 -param2`). Результат выполнения программы будет выглядеть так:

```
Количество элементов: 4
C:\nodejs\node16\node.exe
C:\book\p1\index.js
-param1
-param2
```

Если нужно передать значение, которое содержит пробел, то это значение следует указывать внутри двойных кавычек:

```
C:\book\pl>node index.js x + y "x + y"
```

Количество элементов: 6

```
C:\nodejs\node16\node.exe
```

```
C:\book\pl\index.js
```

```
x
```

```
+
```

```
y
```

```
x + y
```

В первом случае мы получили каждый символ по отдельности, а во втором — все символы вместе, указав значение внутри кавычек.

2.11. Завершение выполнения программы

Программа завершит работу после выполнения самой последней имеющейся в ней инструкции. Однако в некоторых случаях может понадобиться прервать выполнение программы досрочно. Для этого предназначен метод `exit()` объекта `process`. Формат метода:

```
process.exit([<Код завершения>]);
```

Код завершения указывается в виде целого числа: 0 означает нормальное завершение программы, а любое другое число, например 1 — некорректное завершение. Эти числа передаются операционной системе. Если параметр не указан, то используется либо значение 0, либо значение свойства `process.exitCode`. Пример завершения программы с ошибкой:

```
process.exit(1);
```

При выводе сообщения до вызова метода `exit()` следует учитывать, что операции вывода в Node.js могут выполняться асинхронно. Поэтому выводимые данные могут обрезаться или не выводиться вообще. Если вызов метода `exit()` выполнен внутри потока, то завершится только поток, а не основной процесс.

Если нужно только изменить код завершения, достаточно присвоить его свойству `process.exitCode`:

```
process.exitCode = 1;
```

2.12. Получение версии Node.js

Версия Node.js хранится в свойстве `process.version` объекта `process`:

```
console.log(process.version);  
v16.0.0
```

Полная информация доступна через свойство `process.versions` в виде объекта:

```
console.log(process.versions);
```

Результат:

```
{
  node: '16.0.0',
  v8: '9.0.257.17-node.10',
  uv: '1.41.0',
  zlib: '1.2.11',
  brotli: '1.0.9',
  ares: '1.17.1',
  modules: '93',
  nghttp2: '1.42.0',
  napi: '8',
  llhttp: '6.0.0',
  openssl: '1.1.1k+quic',
  cldr: '39.0',
  icu: '69.1',
  tz: '2021a',
  unicode: '13.0',
  ngtcp2: '0.1.0-DEV',
  nghttp3: '0.1.0-DEV'
}
```



ГЛАВА 3

Переменные и типы данных

Переменная — это ячейка памяти, способная хранить одно произвольное значение и имеющая уникальное имя. При занесении в переменную нового значения хранившееся в ней ранее значение теряется.

3.1. Именованние переменных

Имя переменной должно состоять из букв, цифр, знаков доллара, подчеркивания и не может начинаться с цифры. При указании имени переменной важно учитывать регистр букв: `x` и `X` — разные переменные:

```
let x = 5, X = 10;
console.log(x);      // 5
console.log(X);      // 10
```

В качестве имени переменной нельзя использовать ключевые слова: `await`, `break`, `case`, `catch`, `class`, `const`, `continue`, `debugger`, `default`, `delete`, `do`, `else`, `enum`, `export`, `extends`, `false`, `finally`, `for`, `function`, `if`, `import`, `in`, `instanceof`, `new`, `null`, `return`, `super`, `switch`, `this`, `throw`, `true`, `typeof`, `var`, `void`, `while`, `with` и `yield`. Запоминать все ключевые слова нет необходимости, т. к. в редакторе эти слова подсвечиваются. Любая попытка использования ключевого слова в качестве имени переменной приведет к ошибке при запуске. Помимо ключевых слов, следует избегать совпадений имен со встроенными идентификаторами.

❑ Правильные имена переменных: `x`, `y1`, `str_name`, `strName`.

❑ Неправильные имена переменных: `1y`, `str name`, `name1+name2`.

В именах переменных допускается использовать буквы любых алфавитов, например кириллицы:

```
let число = 3;
console.log(число); // 3
```

Однако традиционно применяются лишь буквы латиницы.

Существует несколько стилей именования:

- ❑ Lazy Case — `abc`;
- ❑ Camel Case — `abcAbc`. В языке JavaScript это основной стиль для имен переменных, функций и методов, например: `slice()` и `toLowerCase()`;
- ❑ Pascal Case — `AbcAbc`. В языке JavaScript это основной стиль для имен классов, например: `String` и `WeakMap`;
- ❑ Snake Case — `abc_abc`.

3.2. Объявление переменной

Прежде чем использовать переменную, ее необходимо объявить глобально (вне функций) или локально (внутри функции). Объявление переменной записывается в следующем формате:

```
var|let <Переменная 1>[ = <Значение 1>]
      [, . . . , <Переменная N>[ = <Значение N>]]
```

Пример объявления переменной `x`:

```
let x;
```

В одной инструкции можно объявить сразу несколько переменных, указав их через запятую:

```
let x, y, z;
```

3.3. Инициализация переменной и присваивание значения

При объявлении переменной ей можно присвоить начальное значение, указав его после оператора `=`. Эта операция называется *инициализацией* переменной. Пример:

```
let x, y = 10, z = 30, k;
```

Переменная становится доступной (как говорят программисты, видимой) сразу после объявления, и ее можно использовать для инициализации последующих переменных:

```
let x = 5, y = 10, z = x + y;
console.log(z); // 15
```

Инициализация глобальных (объявленных вне функций) переменных выполняется только один раз. Локальные (объявленные внутри функций) переменные инициализируются при каждом вызове функции.

Присвоить значение переменной можно и после объявления, указав его после оператора `=`. Эта операция называется *присваиванием*. Пример присваивания:

```
let x;
x = 10;
```

Обратите внимание на то, что перед оператором `=` и после него вставлены пробелы. Количество пробелов может быть произвольным, или пробелов может не быть вовсе. Кроме того, допускается вместо пробелов использовать символ перевода строки или символ табуляции. Например, эти инструкции вполне допустимы:

```
let x=21;
let y=           85;
let z
=
56;
```

Тем не менее следует придерживаться единообразия в коде и обрамлять операторы одним пробелом. Таковы рекомендации по оформлению кода.

Если нужно поменять местами значения переменных, то удобно воспользоваться следующим кодом:

```
let x = 5, y = 10;
console.log( x + ' ' + y );    // 5 10
[x, y] = [y, x];
console.log( x + ' ' + y );    // 10 5
```

3.4. Ключевые слова *var* и *let*

Переменные могут быть объявлены с помощью ключевых слов `var` или `let`. Ключевое слово `var` создает глобальную переменную, если объявление расположено вне функции. Пример:

```
var x = 10;
```

В веб-браузере глобальная переменная автоматически становится свойством глобального объекта `window`. В Node.js существование (область видимости) такой переменной ограничено файлом, в котором она объявлена. Но она видна во всех функциях, определенных внутри файла.

Переменная, объявленная посредством ключевого слова `var` внутри блока (например, в первом параметре цикла `for`), будет существовать и после закрывающей фигурной скобки:

```
for (var i = 0; i < 5; i++) {
  console.log(i);
}
console.log('i =', i); // i = 5
```

Результат будет аналогичным при объявлении переменной внутри блока:

```
var x = 5;
if (x == 5) {
  var y = x + 10;
}
console.log('y =', y); // y = 15
```

Указав ключевое слово `var`, одну и ту же переменную можно объявить дважды:

```
var x = 5;
var x = 20;
console.log(x); // 20
```

При попытке обратиться к переменной до ее объявления посредством ключевого слова `var` мы получим значение `undefined`:

```
console.log('x =', x); // x = undefined
var x = 20;
```

Ключевое слово `let` объявляет переменную уровня видимости блока. Если с его помощью переменная объявлена вне блоков, то, как и при использовании ключевого слова `var`, область видимости переменной будет ограничена файлом, в котором она объявлена. Пример:

```
let x = 10;
```

При объявлении с помощью ключевого слова `let` область видимости переменных будет ограничена блоком, в котором переменная объявлена. Переменная будет также видна и во всех вложенных блоках, а вне блока она видна не будет. Пример:

```
let x = 5;
if (x == 5) {
  let y = x + 10;
}
console.log('y =', typeof y); // y = undefined
```

При объявлении переменных в первом параметре цикла `for` область видимости будет ограничена телом цикла:

```
for (let i = 0; i < 5; i++) {
  console.log(i);
}
console.log('i =', typeof i); // i = undefined
```

Повторное объявление переменной посредством ключевого слова `let` приведет к ошибке:

```
{
  let x = 5;
  let x = 20; // SyntaxError: Identifier 'x' has already been declared
}
```

Ошибка также возникнет при обращении к переменной до ее объявления:

```
{
  console.log('x =', x);
  // ReferenceError: Cannot access 'x' before initialization
  let x = 20;
}
```

Если же переменную не объявить вообще и попытаться ее использовать, мы получим ошибку при запуске программы:

```
console.log('x =', x); // ReferenceError: x is not defined
```

Однако если переменной присвоить значение, то в обычном режиме ошибки не будет, хотя мы и не объявляем переменную:

```
x = 10;
console.log('x =', x); // x = 10
```

При использовании строгого режима будет сгенерировано исключение:

```
'use strict';
x = 10; // ReferenceError: x is not defined
```

ПРИМЕЧАНИЕ

Всегда объявляйте переменные посредством ключевого слова `let`. Слово `var` требуется лишь для совместимости со старыми веб-браузерами.

3.5. Типы данных в языке JavaScript

В JavaScript переменные могут хранить значения, относящиеся к следующим типам данных:

- ❑ **boolean** — логический тип данных. Может принимать значения `true` (истина) или `false` (ложь). Пример:

```
console.log(typeof true); // boolean
```

- ❑ **number** — числа, целые и с плавающей точкой (вещественные):

```
console.log(typeof 1); // number
console.log(typeof 2.0); // number
```

- ❑ **bigint** — целые числа больших размеров:

```
console.log(typeof 1n); // bigint
```

- ❑ **string** — строки:

```
console.log(typeof 'строка'); // string
console.log(typeof "строка"); // string
```

- ❑ **symbol** — символьный тип данных (символ) :

```
let id = Symbol('id');
console.log(typeof id); // symbol
```

- ❑ **function** — функции:

```
console.log(typeof function() {}); // function
```

- ❑ **object** — массивы, объекты и значение `null`:

```
console.log(typeof [1, 2, 3]); // object
console.log(typeof {x: 10}); // object
console.log(typeof null); // object
```


- ❑ `undefined` — неопределенное значение (его имеет переменная, которой не было присвоено какое-либо "значимое" значение):

```
let str;  
console.log(typeof str);           // undefined
```

В этих примерах мы воспользовались оператором `typeof`, который возвращает строку, описывающую тип данных переменной. При инициализации переменной JavaScript автоматически относит переменную к одному из типов данных. Пример:

```
let y = 12.5;  
console.log(typeof y);           // number  
console.log(y);                 // 12.5
```

В последующих выражениях тип данных переменной можно изменить:

```
let y = 12.5;  
console.log(typeof y);           // number  
y = 1n;  
console.log(typeof y);           // bigint
```

3.6. Проверка существования переменной

Если в программе обратиться к переменной, которая еще не объявлена, возникнет критическая ошибка. Если переменная объявлена, но ей явно не присвоено какое-либо значение, то значение переменной предполагается равным `undefined`. Пример:

```
let x;  
console.log(x);                 // undefined  
console.log(str);               // ReferenceError: str is not defined
```

Для проверки существования переменной можно воспользоваться оператором `typeof` (ошибка обращения к несуществующей переменной в этом случае не генерируется):

```
if (typeof str == 'undefined') {  
    console.log('Переменная не существует');  
}  
else {  
    console.log('Переменная существует');  
}
```

В этом примере для проверки условия используется оператор ветвления `if`. После названия оператора внутри круглых скобок указывается проверяемое выражение. Если выражение возвращает логическое значение `true`, будут выполнены инструкции внутри фигурных скобок, а блок `else` игнорируется. Если выражение возвращает значение `false`, инструкции внутри фигурных скобок игнорируются, и управление передается блоку `else`. Проверка значения осуществляется с помощью оператора `==` (равно).

3.7. Преобразование типов данных

Попробуем сложить строку и число:

```
let str = '5', x = 3;
let var1 = x + str;      // Переменная содержит строку '35'
console.log(var1);      // 35
console.log(typeof var1); // string
let var2 = str + x;      // Переменная содержит строку '53'
console.log(var2);      // 53
console.log(typeof var2); // string
```

В этой ситуации интерпретатор столкнется с несовместимостью типов данных и попытается сначала преобразовать переменные к одному типу, а затем выполнить операцию. В нашем случае переменная `x`, имеющая тип `number` (число), будет преобразована к типу `string` (строка), а затем будет произведена операция конкатенации строк.

Теперь попробуем из числа вычесть строку, число умножить на строку или число разделить на строку:

```
let str = '5', x = 15;
let var1 = x - str;      // Переменная содержит число 10
console.log(var1);      // 10
console.log(typeof var1); // number
let var2 = x * str;      // Переменная содержит число 75
console.log(var2);      // 75
console.log(typeof var2); // number
let var3 = x / str;      // Переменная содержит число 3
console.log(var3);      // 3
console.log(typeof var3); // number
```

Интерпретатор попытается преобразовать строку в число, а затем вычислить выражение. Причем не важно, в какой последовательности будут указаны число и строка. Пример:

```
let var4 = str * x;      // Переменная содержит число 75
console.log(var4);      // 75
console.log(typeof var4); // number
```

Но что случится, если в строке будут одни буквы?

```
let str = 'Строка', x = 15;
let var1 = x - str;      // Переменная содержит значение NaN
console.log(var1);      // NaN
console.log(typeof var1); // number
```

В этом случае интерпретатор не сможет преобразовать строку в число и присвоит переменной значение `NaN` (*Not a Number*, не число).

С одной стороны, хорошо, что интерпретатор делает преобразование типов данных за нас. Но, с другой стороны, можно получить результат, который вовсе не плани-

ровался. По этой причине лучше оперировать переменными одного типа, а необходимое преобразование типов выполнять явно.

Преобразовать типы данных можно с помощью следующих встроенных функций JavaScript:

- `parseInt(<Строка>[, <Основание>])` — преобразует заданную строку в целое число. Строка считается указанной в системе счисления с заданным основанием. Если основание не указано, система счисления выбирается автоматически и зависит от префиксов в строке (например, если строка начинается с `0x`, будет использоваться основание 16). Если строка не может быть преобразована в число, возвращается значение `NaN`. Пример:

```
console.log( parseInt('10') );           // 10
console.log( parseInt('0xFF') );         // 255
console.log( parseInt('0xFF', 16) );     // 255
console.log( parseInt('0167', 8) );      // 119
console.log( parseInt('1110111', 2) );   // 119
console.log( parseInt('строка', 2) );    // NaN
```

- `parseFloat(<Строка>)` — преобразует строку в число с плавающей точкой:

```
console.log( parseFloat('5.2') );        // 5.2
console.log( parseFloat('5e-2') );       // 0.05
console.log( parseFloat('строка') );     // NaN
```

Для преобразования числа в строку служит метод `toString([<Основание>])`. Если основание не указано, полученное число будет представлено в десятичной системе счисления. Пример:

```
let x = 10, y = 5.2;
console.log( x.toString() );             // 10
console.log( x.toString(8) );            // 12
console.log( y.toString() );             // 5.2
```

Рассмотрим пример использования преобразования типов данных. Просуммируем два числа, переданных пользователем в командной строке (листинг 3.1).

Листинг 3.1. Вычисление суммы двух чисел

```
// Проверяем передачу чисел в командной строке
if (process.argv.length == 4) {
  let x, y, sum1, sum2, msg;
  // Получаем значения из командной строки
  x = process.argv[2];
  y = process.argv[3];
  // Вычисляем сумму чисел
  sum1 = x + y;
  msg = 'До преобразования типов:\n';
  msg += 'Значение суммы чисел ';
  msg += x + ' и ' + y + ' равно ';
```

```
msg += sum1 + '\n\n';
sum2 = parseInt(x, 10) + parseInt(y, 10);
msg += 'После преобразования типов:\n';
msg += 'Значение суммы чисел ' + x + ' и ';
msg += y + ' равно ' + sum2;
console.log(msg);
}
else {
    // Если числа не переданы
    console.log('Передайте 2 числа в конце команды');
}
```

Если мы не укажем числа после команды, то получим сообщение из блока `else`:

```
C:\book\pl>node index.js
Передайте 2 числа в конце команды
```

Если указать два числа 5, в консоли отобразится следующий текст:

```
C:\book\pl>node index.js 5 5
До преобразования типов:
Значение суммы чисел 5 и 5 равно 55
```

```
После преобразования типов:
Значение суммы чисел 5 и 5 равно 10
```

Данные, передаваемые в командной строке или получаемые из диалоговых окон в веб-браузере, имеют тип данных `string` (строка). Чтобы получить сумму двух чисел, их нужно преобразовать в числовой тип, иначе в результате мы получим еще одну строку, а не сумму.

3.8. Классы-"обертки" над элементарными типами

Вы заметили, как мы преобразовывали число в строку в предыдущем разделе?

```
let x = 10, y = 5.2;
console.log( x.toString() );    // 10
```

Мы вызвали метод `toString()` у числа, которое является элементарным типом. В языке JavaScript над элементарными типами данных существуют классы-"обертки". При попытке вызова метода у значения элементарного типа оно автоматически преобразуется в объект соответствующего класса. Так, при попытке вызвать метод `toString()` у числа последнее преобразуется в объект класса `Number`, и упомянутый ранее метод вызывается у полученного объекта.

Объекты классов-"оберток" полностью аналогичны значениям соответствующих элементарных типов и могут участвовать в операциях с этими значениями:

```
var x = new Number(10);
var y = 255;
```

```
console.log(x + y);           // 265
console.log(typeof (x + y));  // number
```

Вот основные классы-"обертки" элементарных типов:

❑ Boolean — тип boolean:

```
var a = new Boolean(true);
var b = a.valueOf();
console.log(typeof b);       // boolean
```

❑ Number — тип number:

```
var x = new Number(10);
var y = x.valueOf();
console.log(typeof y);       // number
```

❑ String — тип string:

```
var a = new String('Строка');
var b = a.valueOf();
console.log(typeof b);       // string
```

3.9. Константы

Константы служат для хранения значений, которые не должны изменяться во время работы программы. Создать константу можно с помощью ключевого слова `const`:

```
const <Имя константы> = <Значение константы>
```

Значение можно присвоить константе только при ее объявлении. После объявления изменить значение константы нельзя. Пример:

```
const MY_CONST = 10;
console.log( MY_CONST );           // 10
```

НА ЗАМЕТКУ

В JavaScript хорошим тоном считается повсеместное использование ключевого слова `const` вместо `var` и `let`, если значение не изменяется в программе.

Как и при объявлении переменной, в одной инструкции можно объявить несколько констант через запятую:

```
const MY_CONST1 = 10, MY_CONST2 = 5, MY_CONST3 = 88;
```

Если константе присвоен объект, то заменить этот объект другим нельзя, а вот изменить свойства объекта можно:

```
const MY_CONST = [1, 2, 3];
console.log( MY_CONST );           // [ 1, 2, 3 ]
MY_CONST[0] = 88;
console.log( MY_CONST );           // [ 88, 2, 3 ]
```

3.10. Области видимости переменных

Объявить переменную можно глобально (вне функций) или локально (внутри функций или блока).

ВНИМАНИЕ!

Сказанное в этом параграфе также относится и к константам.

- ❑ *Глобальные переменные* — объявленные в файле вне функций. Видны в любой части файла, в котором они объявлены, включая функции.
- ❑ *Локальные переменные* — объявленные внутри функций. Видны только внутри функции. Инициализируются при каждом вызове функции, после выхода из функции уничтожаются.
- ❑ *Локальные переменные внутри блока* — объявленные внутри неименованного блока (в области, ограниченной фигурными скобками) с помощью ключевого слова `let` или `const`. Видны только внутри блока и во вложенных в него блоках. Инициализируются при каждом входе в блок, после выхода из блока уничтожаются.

Если имя локальной переменной совпадает с именем глобальной переменной, то все операции внутри функции осуществляются с локальной переменной, а значение глобальной не изменяется.

Область видимости глобальных и локальных переменных приведена в листинге 3.2.

Листинг 3.2. Область видимости переменных

```
function func() {  
    var n = 10;           // Локальная переменная  
    let x = 60;           // Локальная переменная  
    const LY = 7;         // Локальная константа  
    // Доступ к глобальным переменным и константам  
    console.log(g1);      // 5  
    console.log(GX);      // 8  
    // Можно изменить значение глобальной переменной  
    g1 = 33;  
    // Доступ к локальным переменным и константам  
    console.log(n);       // 10  
    console.log(x);       // 60, а не 3  
    console.log(LY);      // 7  
    // Изменяется локальная переменная  
    x = 88;  
}  
  
var g1 = 5;               // Глобальная переменная  
let x = 3;                // Глобальная переменная  
const GX = 8;            // Глобальная константа
```

```
// Вызываем функцию
func();

console.log(g1);           // 33 (значение изменилось)
console.log(x);            // 3  (значение не изменилось)
console.log(GX);           // 8
```

Не забываем, что переменная, объявленная внутри блока с помощью ключевого слова `let` или `const`, видна только в пределах блока (внутри фигурных скобок). Например, присвоим значение переменной в зависимости от некоторого условия:

```
// Неправильно!!!
let x = 10;
if (x == 10) { // Какое-то условие
    let y;
    y = 5;
    console.log(y);
}
else {
    let y;
    y = 25;
    console.log(y);
}
// console.log(y); // Ошибка! Переменная y здесь не видна!
// ReferenceError: y is not defined
```

В этом примере переменная `y` видна только внутри блока. После условного оператора `if` переменной не существует. Чтобы переменная была видна внутри блока и после выхода из него, необходимо поместить объявление переменной перед блоком:

```
// Правильно
let x = 10;
let y = 0;
if (x == 10) { // Какое-то условие
    y = 5;
    console.log(y);
}
else {
    y = 25;
    console.log(y);
}
console.log(y);
```

Язык JavaScript также поддерживает *неименованные блоки*. Такие блоки не привязаны ни к какой инструкции — просто фигурные скобки сами по себе. Если переменная объявлена внутри такого блока с помощью ключевого слова `let`, то после блока она уже не будет видна. Точно такая же ситуация при объявлении констант.

Пример:

```
{ // Блок
  var x = 3;
  let y = 25;
  const Z = 10;
}
console.log(x);    // 3. OK
// console.log(y); // Ошибка! Переменная y здесь не видна!
// ReferenceError: y is not defined
// console.log(Z); // Ошибка! Константа Z здесь не видна!
// ReferenceError: Z is not defined
```

Заметьте, что к переменной, объявленной с помощью ключевого слова `var`, это правило не применяется.

Неименованные блоки могут быть вложенными. Если во вложенном блоке объявлена одноименная переменная с помощью ключевого слова `let`, то создается новая переменная. Если указано ключевое слово `var`, то значение переменной будет перезаписано. Пример:

```
var x = 1;
let y = 2;
{ // Блок
  var x = 3;
  let y = 25;
  console.log(x);    // 3
  console.log(y);    // 25
  { // Блок
    var x = 16;
    let y = 77;
    console.log(x); // 16
    console.log(y); // 77
  }
}
console.log(x);      // 16 (Значение изменилось!)
console.log(y);      // 2  (Значение не изменилось!)
```

В цикле `for` переменная, объявленная внутри круглых скобок с помощью ключевого слова `let`, видна только внутри блока:

```
for (let i = 0; i < 10; i++) {
  console.log(i);
}
// console.log(i); // Ошибка! Переменная i здесь не видна!
// ReferenceError: i is not defined
```

Если переменную объявить с помощью ключевого слова `var`, она будет видна после выхода из блока:


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

3.11. Вывод значений переменных

Вывести значение переменной любого типа позволяет метод `log()` встроенного объекта `console`:

```
let x = 10;  
console.log(x); // 10
```

Если нужно предварительно вывести описание, то в первом параметре метода `log()` указывается строка с описанием, а во втором — переменная:

```
let x = 10;  
console.log('x = ', x); // x = 10
```

Можно воспользоваться операцией конкатенации строк. В этом случае после строки с описанием указывается оператор `+`, за которым следует переменная. Пример:

```
let x = 10;  
console.log('x = ' + x); // x = 10
```

Еще один способ заключается в использовании *шаблонов строк*. Строка с описанием указывается внутри обратных апострофов. Переменная подставляется в такую строку следующим образом:

```
${<Переменная>}
```

Вначале идет символ `$`, за которым внутри фигурных скобок указывается имя переменной:

```
let x = 10;  
console.log(`x = ${x}`); // x = 10
```



ГЛАВА 4

Операторы и циклы

Операторы выполняют элементарные действия с данными. Например, операторы присваивания сохраняют данные в переменных, математические операторы производят арифметические вычисления, а условные операторы в зависимости от значения логического выражения выполняют или, наоборот, не выполняют заданный набор инструкций.

4.1. Математические операторы

Математические операторы, доступные в JavaScript:

□ + — сложение:

```
console.log( 10 + 15 );           // 25
```

□ - — вычитание:

```
console.log( 30 - 15 );           // 15
```

□ * — умножение:

```
console.log( 12 * 10 );           // 120
```

□ / — деление:

```
console.log( 10 / 2 );             // 5
console.log( 10 / 3 );             // 3.3333333333333335
```

Деление числа на 0 приведет к значению плюс или минус Infinity (бесконечность), а деление числа 0 на 0 — к значению NaN (нет числа):

```
console.log( 10 / 0 );             // Infinity
console.log( -10 / 0 );            // -Infinity
console.log( 0 / 0 );             // NaN
```

□ % — остаток от деления:

```
console.log( 10 % 2 );             // 0 (10 - 10 / 2 * 2)
console.log( 10 % 3 );             // 1 (10 - 10 / 3 * 3)
console.log( 10 % 4 );             // 2 (10 - 10 / 4 * 4)
console.log( 10 % 6 );             // 4 (10 - 10 / 6 * 6)
```

❑ **** — возведение в степень:**

```
console.log( 10 ** 2 );           // 100
console.log( 3 ** 3 );           // 27
console.log( -(3 ** 3) );        // -27
console.log( Math.pow(10, 2) );  // 100
```

❑ **- — унарный минус:**

```
let x = 10;
console.log( -x );               // -10
```

❑ **+ — унарный плюс. Преобразует значение в числовой тип (number). Пример:**

```
let n = '10';
let x = +n;
console.log( x );               // 10
console.log( typeof x );       // number
```

❑ **++ — оператор инкремента. Увеличивает значение переменной на 1. Пример:**

```
let x = 10;
x++;                             // Эквивалентно x = x + 1;
console.log( x );               // 11
```

❑ **-- — оператор декремента. Уменьшает значение переменной на 1. Пример:**

```
let x = 10;
x--;                             // Эквивалентно x = x - 1;
console.log( x );               // 9
```

Операторы инкремента и декремента могут использоваться в *постфиксной* или *префиксной* форме:

```
x++; x--; // Постфиксная форма
++x; --x; // Префиксная форма
```

При постфиксной форме (x++) сначала возвращается значение переменной, а уже потом выполняется операция, а при префиксной форме (++x) — вначале выполняется операция, и только потом возвращается значение переменной. Продемонстрируем разницу на примере (листинг 4.1).

Листинг 4.1. Постфиксная и префиксная форма

```
let x = 5, y, msg;
y = x++;           // y = 5, x = 6
msg = 'Постфиксная форма (y = x++):\n';
msg += 'y = ' + y + '\n';
msg += 'x = ' + x + '\n';
x = 5;
y = ++x;           // y = 6, x = 6
msg += 'Префиксная форма (y = ++x):\n';
msg += 'y = ' + y + '\n';
```

```
msg += 'x = ' + x;  
console.log(msg);
```

В итоге получим следующий результат:

Постфиксная форма ($y = x++$):

```
y = 5
```

```
x = 6
```

Префиксная форма ($y = ++x$):

```
y = 6
```

```
x = 6
```

Если операторы инкремента и декремента присутствуют в сложных выражениях, то понять, каким будет результат выполнения выражения, становится сложно. Например, каким будет значение переменной y после выполнения инструкций, приведенных далее?

```
let x = 5, y = 0;
```

```
y = ++x + ++x;
```

Чтобы облегчить жизнь себе и другим программистам, операторы инкремента и декремента лучше использовать отдельно от других операторов.

4.2. Побитовые операторы

Побитовые операторы выполняют поразрядные действия с двоичным представлением целых чисел. Перечислим побитовые операторы, доступные в JavaScript:

□ $\sim$ — двоичная инверсия. Значение каждого бита заменяется на противоположное. Пример:

```
y = ~x;
```

□ $\&$ — двоичное И:

```
let x = 100, y = 75, z;  
console.log( x.toString(2) );           // 1100100  
console.log( y.toString(2) );           // 1001011  
z = x & y;  
console.log( z.toString(2) );           // 1000000
```

□ $|$ — двоичное ИЛИ:

```
let x = 100, y = 75, z;  
console.log( x.toString(2) );           // 1100100  
console.log( y.toString(2) );           // 1001011  
z = x | y;  
console.log( z.toString(2) );           // 1101111
```

□ $\wedge$ — двоичное исключающее ИЛИ:

```
let x = 100, y = 250, z;  
console.log( x.toString(2) );           // 01100100
```

```
console.log( y.toString(2) );           // 11111010
z = x ^ y;
console.log( z.toString(2) );           // 10011110
```

- << — сдвиг влево на указанное число разрядов с заполнением младших разрядов нулями:

```
let x = 100;
console.log( x.toString(2) );           // 001100100
x = x << 1;
console.log( x.toString(2) );           // 011001000
x = x << 1;
console.log( x.toString(2) );           // 110010000
```

- >> — сдвиг вправо на указанное число разрядов с заполнением старших разрядов содержимым самого старшего разряда:

```
x = x >> 1;
```

- >>> — сдвиг вправо без учета знака на указанное число разрядов с заполнением старших разрядов нулями:

```
x = x >>> 1;
```

4.3. Оператор запятая

Оператор запятая позволяет разместить сразу несколько выражений внутри одной инструкции. Например, в предыдущих примерах мы использовали этот оператор для объявления нескольких переменных в одной инструкции:

```
let x, y, z;
```

Результат вычисления последнего выражения можно присвоить какой-либо переменной. При этом выражения должны быть расположены внутри круглых скобок, т. к. приоритет оператора запятая меньше приоритета оператора присваивания.

Пример:

```
let x = 0, y = 0, z = 0;
x = (y = 10, z = 20, y + z);
console.log(x);    // 30
console.log(y);    // 10
console.log(z);    // 20
```

В этом примере после объявления и инициализации переменных переменной `y` присваивается значение 10, затем переменной `z` присваивается 20, далее вычисляется выражение `y + z` и его результат присваивается переменной `x`.

4.4. Операторы сравнения

Операторы сравнения используются в логических выражениях. Перечислим их:

□ == — равно;	□ < — меньше;
□ === — строго равно;	□ > — больше;
□ != — не равно;	□ <= — меньше или равно;
□ !== — строго не равно;	□ >= — больше или равно.

Логические выражения возвращают одно из двух значений: `true` (истина) или `false` (ложь):

```
console.log( 10 == 10 ); // true
console.log( 10 == 5  ); // false
console.log( 10 != 5  ); // true
console.log( 10 < 5   ); // false
console.log( 10 > 5   ); // true
console.log( 10 <= 5  ); // false
console.log( 10 >= 5  ); // true
```

Если используется оператор `==`, интерпретатор пытается преобразовать разные типы данных к одному и лишь затем сравнивает их. Оператор `===`, встретив данные разных типов, сразу возвращает `false` (ложь). Пример:

```
console.log( 5 == '5' ); // true
console.log( 5 === '5' ); // false
```

Следует учитывать, что оператор проверки на равенство содержит два символа `=`. Указание одного символа `=` является логической ошибкой, т. к. этот оператор предназначен для присваивания значения переменной, а не для проверки условия. Использовать оператор присваивания внутри логического выражения допускается, поэтому интерпретатор не выведет сообщение об ошибке, однако программа может выполняться некорректно. Например, в следующем примере вместо проверки равенства числу 11 выполняется присваивание:

```
let x = 10;
if (x = 11) {
  console.log('x == 11');
}
```

Выражение внутри круглых скобок присвоит переменной `x` значение 11, а затем будет выполнено сравнение. Любое число, не равное 0, трактуется как логическое значение `true`, поэтому в консоли отобразится строка `x == 11`.

Выполнить сравнение двух значений также позволяет статический метод `is(<Значение 1>, <Значение 2>)` класса `Object`. Метод возвращает значение `true` в следующих случаях:

```
console.log( Object.is(undefined, undefined) ); // true
console.log( Object.is(null, null) );           // true
```

```

console.log( Object.is(true, true) );           // true
console.log( Object.is(false, false) );         // true
console.log( Object.is('abc', 'abc') );         // true
let obj1 = { a: 10 }, obj2 = obj1;
console.log( Object.is(obj1, obj2) );           // true
console.log( Object.is(+0, +0) );               // true
console.log( Object.is(-0, -0) );               // true
console.log( Object.is(NaN, NaN) );             // true
console.log( Object.is(1, 1) );                 // true

```

Метод `is()` не выполняет приведение разных типов, поэтому его действие отличается от оператора `==`. Имеются также отличия от оператора `===`:

```

console.log( NaN === NaN );                     // false
console.log( Object.is(NaN, NaN) );             // true
console.log( +0 === -0 );                       // true
console.log( Object.is(+0, -0) );               // false

```

Значение логического выражения можно инвертировать с помощью оператора `!` таким образом:

```

console.log( 10 == 10 );                       // true
console.log( !(10 == 10) );                    // false

```

Поскольку значения равны, должно быть возвращено значение `true`, но т. к. перед выражением стоит оператор `!`, выражение вернет `false`.

Несколько логических выражений можно объединить в одно большое с помощью следующих операторов:

□ **&& — логическое И.** Логическое выражение вернет `true` только в случае, если оба подвыражения вернут `true`. Пример:

```

console.log( (10 == 10) && (5 != 3) );          // true
console.log( (10 == 10) && (5 == 3) );          // false

```

□ **|| — логическое ИЛИ.** Логическое выражение вернет `true`, если хотя бы одно из подвыражений вернет `true`. Если первое подвыражение вернет значение `true`, второе подвыражение даже не будет вычисляться. Пример:

```

console.log( (10 == 10) || (5 != 3) );          // true
console.log( (10 == 10) || (5 == 3) );          // true

```

Оператор `||` ранее часто применялся для создания необязательных параметров в функции. Если первое выражение не может быть преобразовано в `true`, то возвращается значение второго выражения. Пример:

```

function test(str) {
  str = str || 'Значение по умолчанию';
  return str;
}

console.log( test() );                          // Значение по умолчанию
console.log( test('Значение указано') );        // Значение указано

```

Если оператор `!` указать дважды, то любое значение преобразуется в логическое:

```
console.log( !!10 );           // true
```

Еще для преобразования любого значения в логическое можно использовать класс `Boolean`:

```
let a = new Boolean(10);
console.log(a);                // [Boolean: true]
console.log( typeof a );       // object
console.log( a.valueOf() );     // true
console.log( typeof a.valueOf() ); // boolean
let b = Boolean(10);
console.log(b);                // true
console.log( typeof b );       // boolean
```

4.5. Оператор нулевого слияния

Оператор нулевого слияния `??` возвращает значение левого подвыражения, если оно не равно `undefined` или `null`, в противном случае возвращает значение правого подвыражения:

```
console.log(321 ?? 123);       // 321
console.log(undefined ?? 123); // 123
```

4.6. Операторы присваивания

Операторы присваивания заносят значения в переменные. Вот операторы присваивания, доступные в JavaScript:

□ `=` — присваивает значение переменной:

```
let x = 5;
```

Оператор `=` можно также использовать для распаковки объектов:

```
let arr = [1, 2];
let [a, b] = arr;
console.log( a + ' ' + b ); // 1 2
let obj = { c: 3, d: 4 };
let { c, d } = obj;
console.log( c + ' ' + d ); // 3 4
```

□ `+=` — увеличивает значение переменной на указанную величину:

```
x += 5; // Эквивалентно x = x + 5;
```

□ `-=` — уменьшает значение переменной на указанную величину:

```
x -= 5; // Эквивалентно x = x - 5;
```

□ `*=` — умножает значение переменной на указанную величину:

```
x *= 5; // Эквивалентно x = x * 5;
```


□ `/=` — делит значение переменной на указанную величину:

```
x /= 5; // Эквивалентно x = x / 5;
```

□ `%=` — делит значение переменной на указанную величину и возвращает остаток:

```
x %= 5; // Эквивалентно x = x % 5;
```

□ `**=` — возведение в степень с присваиванием:

```
let x = 10;
x **= 2; // Эквивалентно x = x ** 2;
console.log(x); // 100
```

□ `&=`, `|=`, `^=`, `<<=`, `>>=` и `>>>=` — побитовые операторы с присваиванием;

□ `&&=` и `||=` — логические И и ИЛИ с присваиванием;

□ `??=` — нулевое слияние с присваиванием:

```
let a = 10, b;
console.log(a); // 10
a ??= 100;
console.log(a); // 10
console.log(b); // undefined
b ??= 100;
console.log(b); // 100
```

4.7. Приоритет выполнения операторов

В какой последовательности будет вычисляться это выражение?

```
let x = 5 + 10 * 3 / 2;
console.log( x ); // 20
```

Это зависит от приоритета выполнения операторов. В данном случае последовательность вычисления выражения будет следующей:

1. Число 10 будет умножено на 3, т. к. приоритет оператора умножения выше приоритета оператора сложения.
2. Полученное значение будет поделено на 2, поскольку приоритет оператора деления равен приоритету оператора умножения (а операторы с равными приоритетами выполняются слева направо), но выше, чем у оператора сложения.
3. К полученному значению будет прибавлено число 5.
4. Значение будет присвоено переменной `x`, т. к. оператор присваивания имеет наименьший приоритет.

С помощью скобок можно изменить последовательность вычисления выражения:

```
let x = (5 + 10) * 3 / 2;
console.log( x ); // 22.5
```

Теперь порядок вычислений окажется другим:

1. К числу 5 будет прибавлено 10.
2. Полученное значение будет умножено на 3.
3. Полученное значение будет поделено на 2.
4. Значение будет присвоено переменной *x*.

Перечислим операторы в порядке убывания приоритета:

1. ++, -- — постфиксный инкремент, постфиксный декремент;
2. !, ~, ++, --, +, - — отрицание, двоичная инверсия, префиксный инкремент, префиксный декремент, унарный плюс, унарный минус;
3. ** — возведение в степень;
4. *, /, % — умножение, деление, остаток от деления;
5. +, - — сложение и вычитание;
6. <<, >>, >>> — двоичные сдвиги;
7. <, <=, >, >= — операторы сравнения;
8. ==, !=, ===, !== — операторы сравнения;
9. & — двоичное И;
10. ^ — двоичное исключающее ИЛИ;
11. | — двоичное ИЛИ;
12. ?? — нулевое слияние;
13. =, +=, -=, *=, /=, %=, **=, &=, |=, ^=, <<=, >>=, >>>=, &&=, ||= и ??= — присваивание;
14. , — запятая.

4.8. Оператор ветвления *if...else*

Условные операторы позволяют в зависимости от значения логического выражения выполнить или, наоборот, не выполнять заданный набор инструкций.

Оператор ветвления *if...else* имеет следующий формат:

```
if (<Логическое выражение>) {  
    <Блок, выполняемый, если условие истинно>  
}  
[else {  
    <Блок, выполняемый, если условие ложно>  
}]
```

Если логическое выражение возвращает значение *true*, выполняются инструкции, расположенные внутри фигурных скобок сразу после оператора *if*. Если логическое выражение возвращает значение *false*, выполняются инструкции после ключевого слова *else*. Блок *else* является необязательным. Допускается не указывать

фигурные скобки, если блоки состоят из одной инструкции. В качестве примера проверим число, введенное пользователем в командной строке, на четность и выведем соответствующее сообщение (листинг 4.2).

Листинг 4.2. Проверка числа на четность

```
// Проверяем передачу числа в командной строке
if (process.argv.length == 3) {
  let x = parseInt(process.argv[2], 10);
  if ( isNaN(x) ) {
    console.log('Вы ввели не число!');
  }
  else {
    if ( (x % 2) == 0 ) {
      console.log('Четное число');
    }
    else {
      console.log('Нечетное число');
    }
  }
}
else {
  // Если число не передано
  console.log('Передайте число в конце команды');
}
```

Первое условие проверяет наличие числа в составе команды. Если число элементов массива `process.argv` не равно трем, выполняется последний блок `else`, внутри которого выводится подсказка пользователю:

```
C:\book\pl>node index.js
Передайте число в конце команды
```

Первый элемент массива `process.argv` содержит путь к программе `node.exe`, а второй — путь к исполняемому файлу. Следовательно, при передаче числа массив должен состоять из трех элементов.

Получаем из массива `process.argv` элемент с индексом 2 (не забываем, что нумерация элементов массива начинается с 0). Полученное значение, имеющее тип `string` (строка), преобразуем в число с помощью функции `parseInt()`. Мы ожидаем десятичное значение, поэтому во втором параметре указываем основание 10.

Вместо числа пользователь может ввести что угодно. Поэтому, прежде чем проверять число на четность, нужно убедиться, что преобразование типов выполнено успешно. Если преобразовать строку в число невозможно, функция `parseInt()` вернет значение `NaN` (нет числа). Велик соблазн сравнить значение переменной `x` с `NaN`, но тут мы столкнемся с проблемой. Значение `NaN` не равно даже самому себе. Пример:

```
let y = NaN;
console.log( NaN == NaN );           // false
console.log( NaN === NaN );          // false
console.log( y === NaN );             // false
```

Чтобы проверить на равенство значению NaN, следует использовать функцию `isNaN()`, которая возвращает `true`, если значение, указанное в качестве параметра, равно NaN, и `false` — в противном случае:

```
let a = NaN, b = 5;
console.log( isNaN(a) === true );     // true
console.log( isNaN(b) === true );     // false
```

Проверка на равенство выражения значению `true` (истина) выполняется по умолчанию, поэтому можно просто написать так:

```
let a = NaN, b = 5;
console.log( isNaN(a) );               // true
console.log( isNaN(b) );               // false
```

Именно так мы и сделали в листинге 4.2. Если пользователь ввел не число, то выводим соответствующее сообщение, а блок `else` при этом пропускается:

```
C:\book\pl>node index.js str
Вы ввели не число!
```

Если пользователь ввел число, управление передается блоку `else`, внутри которого мы опять проверяем условие. Если деление по модулю (оператор `%`) на 2 выполняется без остатка (результат такого деления равен нулю), то число является четным:

```
C:\book\pl>node index.js 10
Четное число
```

В противном случае число является нечетным и выполняется блок `else`:

```
C:\book\pl>node index.js 11
Нечетное число
```

В качестве логического выражения можно указать просто значение, которое будет преобразовано в `false` в следующих случаях:

❑ если число равно 0 или 0.0:

```
console.log( !0 );                     // false
console.log( !0.0 );                   // false
```

❑ если указана пустая строка:

```
console.log( !"" );                    // false
```

❑ если переменная содержит значение `null`:

```
console.log( !null );                  // false
```

❑ если переменная содержит значение `NaN`:

```
console.log( !NaN );                   // false
```

❑ если свойство объекта не существует, или переменной не присвоено начальное значение:

```
let obj = {}, x;
console.log( !!obj.attr ); // false
console.log( !!x );        // false
console.log( !!undefined ); // false
```

Все остальные значения будут преобразованы в true:

```
console.log( !!10 );           // true
console.log( !!"0" );         // true
let obj1 = {}, obj2 = { n: 20 };
console.log( !!obj1 );        // true
console.log( !!obj2.n );      // true
```

Как видно из листинга 4.2, один условный оператор можно вложить в другой. Кроме того, если блок состоит из одной инструкции, фигурные скобки можно не указывать. Пример:

```
if ( (x % 2) == 0 ) console.log('Четное число');
else console.log('Нечетное число');
```

Более того, блока else может не быть совсем:

```
if ( (x % 2) == 0 ) console.log('Четное число');
```

Инструкции в условном операторе можно записать так:

```
let x = 10;
if ( (x % 2) == 0 )
    console.log('Четное число');
else
    console.log('Нечетное число');
console.log('x = ' + x);
```

Если фигурные скобки не указаны, то считается, что внутри блока содержится только одна инструкция. Поэтому последняя инструкция к блоку else не относится. Она будет выполнена в любом случае, вне зависимости от условия.

Ради наглядности перед инструкциями, расположенными внутри блока, добавлено одинаковое количество пробелов. Если записать код следующим образом, ничего не изменится:

```
let x = 10;
if ( (x % 2) == 0 )
    console.log('Четное число');
else
    console.log('Нечетное число');
console.log('x = ' + x);
```

Однако в дальнейшем разбираться в таком коде будет неудобно. Поэтому перед инструкциями внутри блока всегда следует размещать одинаковый отступ. В качестве отступа можно использовать пробелы или символ табуляции. В случае пробе-

лов размер отступа равняется трем или четырем пробелам для блока первого уровня. Для вложенных блоков количество пробелов умножают на уровень вложенности: если для создания отступов у блока первого уровня вложенности использовались три пробела, то для создания отступов у блока второго уровня вложенности следует применить шесть пробелов, для отступа у блока третьего уровня — девять пробелов и т. д. Также не следует комбинировать в отступах пробелы и табуляцию — нужно выбрать что-то одно.

При отсутствии пробелов или фигурных скобок чтение программы может вызывать затруднения. Даже если известен приоритет выполнения операторов, всегда закрадывается сомнение и чтение программы останавливается. Например, непонятно, к какому оператору `if` принадлежит блок `else` в следующем коде:

```
if (x >= 0) if (x == 0) y = 0; else y = 1;
```

В данном случае следует расставить фигурные скобки:

```
if (x >= 0) { if (x == 0) y = 0; else y = 1; }
```

А если сделать так, то чтение и понимание программы станет мгновенным:

```
if (x >= 0) {  
    if (x == 0) y = 0;  
    else y = 1;  
}
```

Если блок состоит из нескольких инструкций, следует указать фигурные скобки. Существует несколько стилей размещения скобок в операторе `if`:

```
// Стил ь 1  
if (<Логическое выражение>) {  
    // Инструкции  
}  
else {  
    // Инструкции  
}  
  
// Стил ь 2  
if (<Логическое выражение>) {  
    // Инструкции  
} else {  
    // Инструкции  
}  
  
// Стил ь 3  
if (<Логическое выражение>)  
{  
    // Инструкции  
}  
else  
{  
    // Инструкции  
}
```

Многие программисты считают стиль 3 наиболее приемлемым, т. к. открывающая и закрывающая скобки расположены друг под другом. Однако, на взгляд авторов, образуются лишние пустые строки. Так как размеры экрана ограничены, при наличии пустой строки на экран помещается меньше кода и приходится чаще пользоваться полосой прокрутки. Если размещать инструкции с равным отступом, то блок кода выделяется визуально и следить за положением фигурных скобок просто излишне (тем более что редактор VS Code подсвечивает парные скобки). Какой стиль выбрать, зависит от личного предпочтения программиста или от правил оформления кода, принятых в определенной фирме. Главное, чтобы стиль оформления внутри одной программы был одинаковым. В этой книге мы будем пользоваться стилем 1.

Для проверки нескольких условий часто используется следующий синтаксис:

```
if (<Условие 1>) {  
    <Блок 1>  
}  
else if (<Условие 2>) {  
    <Блок 2>  
}  
// . . . Фрагмент опущен . . .  
else if (<Условие N>) {  
    <Блок N>  
}  
else {  
    <Блок else>  
}
```

Если <Условие 1> истинно, выполняется <Блок 1>, а все остальные условия пропускаются. Если <Условие 1> ложно, проверяется <Условие 2>. Если <Условие 2> истинно, выполняется <Блок 2>, а все остальные условия пропускаются. Если <Условие 2> ложно, то точно так же проверяются остальные условия. Если все условия ложны, выполняется <Блок else>. В качестве примера определим, какое число от 0 до 2 ввел пользователь (листинг 4.3).

Листинг 4.3. Проверка нескольких условий

```
// Проверяем передачу числа в командной строке  
if (process.argv.length == 3) {  
    let x = parseInt(process.argv[2], 10);  
    if ( isNaN(x) ) {  
        console.log('Вы ввели не число!');  
    }  
    else {  
        if (x == 0)  
            console.log('Вы ввели число 0');  
        else if (x == 1)  
            console.log('Вы ввели число 1');
```

```

    else if (x == 2)
        console.log('Вы ввели число 2');
    else {
        console.log('Вы ввели другое число');
        console.log(`x = ${x}`);
    }
}

}

else {
    // Если число не передано
    console.log('Передайте число от 0 до 2 в конце команды');
}
}

```


Если <Логическое выражение> возвращает значение `true`, выполняется выражение, расположенное после вопросительного знака. Если <Логическое выражение> возвращает `false`, выполняется выражение, расположенное после двоеточия. Результат выполнения выражений становится результатом выполнения оператора. Пример проверки числа на четность и вывода результата:

```
let x = 10;
console.log( (x % 2 == 0) ? `${x} - четное число` :
              `${x} - нечетное число` );
```

Обратите внимание на то, что в качестве операндов указываются именно выражения, а не инструкции. Кроме того, выражения обязательно должны возвращать какое-либо значение. Так как оператор возвращает значение, его можно вставлять внутрь выражений. Пример:

```
let x, y;
x = 0;
y = 30 + 10 / (!x ? 1 : x); // 30 + 10 / 1
console.log(y);           // 40
x = 2;
y = 30 + 10 / (!x ? 1 : x); // 30 + 10 / 2
console.log(y);           // 35
```

В качестве операнда можно указать функцию, которая возвращает значение:

```
function func1(x) {
  console.log(`${x} - четное число`);
  return 0;
}
function func2(x) {
  console.log(`${x} - нечетное число`);
  return 0;
}
let x = 11;
(x % 2 == 0) ? func1(x) : func2(x);
```

Как видно из примера, значение, возвращаемое оператором, можно проигнорировать.

4.10. Оператор выбора *switch*

Оператор выбора `switch` имеет следующий формат:

```
switch (<Переменная или выражение>) {
  case <Значение 1>:
    <Инструкции>
    break;
  [ case <Значение 2>:
    <Инструкции>
    break;
  ...]
```

```
[ default:
    <Инструкции>
]
```

Вместо условия оператор `switch` принимает переменную или выражение. В зависимости от его значения выполняется один из блоков `case`, в котором указано это значение. Если ни одно из значений не описано в блоках `case`, выполняется блок `default` (если он указан). Сравнение осуществляется с помощью оператора `===` (строго равно). Пример использования оператора `switch` приведен в листинге 4.4.

Листинг 4.4. Использование оператора `switch`

```
if (process.argv.length == 3) {
    let os = parseInt(process.argv[2], 10);
    if ( isNaN(os) ) {
        console.log('Вы ввели не число!');
    }
    else {
        switch (os) {
            case 1:
                console.log('Вы выбрали - Windows 7');
                break;
            case 2:
                console.log('Вы выбрали - Windows 8');
                break;
            case 3:
                console.log('Вы выбрали - Windows 10');
                break;
            case 4:
                console.log('Вы выбрали - Другая');
                break;
            default:
                console.log('Мы не смогли определить систему');
        }
    }
}
else {
    console.log('Какой операционной системой вы пользуетесь?');
    console.log('1 - Windows 7');
    console.log('2 - Windows 8');
    console.log('3 - Windows 10');
    console.log('4 - Другая\n');
    console.log(
        'Передайте число, соответствующее ответу, в конце команды');
}
```

Пример ввода значений:

```
C:\book\pl>node index.js
```

Какой операционной системой вы пользуетесь?

- 1 - Windows 7
- 2 - Windows 8
- 3 - Windows 10
- 4 - Другая

Передайте число, соответствующее ответу, в конце команды

```
C:\book\pl>node index.js str
```

Вы ввели не число!

```
C:\book\pl>node index.js 3
```

Вы выбрали - Windows 10

```
C:\book\pl>node index.js 4
```

Вы выбрали - Другая

```
C:\book\pl>node index.js 5
```

Мы не смогли определить систему

Как видно из примера, в конце каждого блока `case` присутствует оператор `break`. Этот оператор позволяет досрочно выйти из оператора выбора `switch`. Если его опустить, то будет выполняться следующий блок `case` вне зависимости от указанного значения. Иными словами, оператор `break` следует обязательно указывать в конце каждого блока `case`. В некоторых случаях это может быть полезным. Например, можно выполнить одни и те же инструкции при разных значениях, поместив инструкции в конце диапазона значений. Пример:

```
let ch = 'b';
switch (ch) {
  case 'a':
  case 'b':
  case 'c':
    console.log('a, b или c');
    break;
  case 'd':
    console.log('Только d');
    break;
}
```

Для быстрой вставки оператора `switch` в редакторе VS Code предназначена следующая комбинация:

```
switch =>
switch (key) {
  case value:

    break;
```

```
    default:
        break;
}
```

4.11. Цикл *for*

Циклы позволяют выполнить одни и те же инструкции многократно. Предположим, нужно вывести все числа от 1 до 100 по одному на строке. Обычным способом пришлось бы писать 100 строк кода:

```
console.log('1');
console.log('2');
// ...
console.log('100');
```

При помощи циклов то же действие можно выполнить одной строкой кода:

```
for (let i = 1; i < 101; i++) console.log(`${i}`);
```

Цикл *for* предназначен для выполнения инструкций заданное число раз. Цикл имеет следующий формат:

```
for (<Начальное значение>; <Условие>; <Приращение>) {
    <Инструкции>
}
```

Параметры имеют следующие значения:

- <Начальное значение> — присваивает переменной-счетчику начальное значение;
- <Условие> — содержит логическое выражение. Пока оно возвращает значение *true*, выполняются инструкции внутри цикла. Обратите внимание, что <Условие> вычисляется на каждой итерации цикла;
- <Приращение> — задает изменение переменной-счетчика при каждой итерации.

Последовательность работы цикла *for*:

1. Переменной-счетчику присваивается начальное значение.
2. Проверяется условие; если оно истинно, выполняются инструкции внутри цикла, в противном случае выполнение цикла завершается.
3. Переменная-счетчик изменяется на величину, указанную в параметре <Приращение>.
4. Переход к п. 2.

Переменная-счетчик может быть объявлена как вне цикла *for*, так и в параметре <Начальное значение>. Если переменная объявлена в параметре с помощью ключевого слова *let*, она будет видна только внутри цикла. Кроме того, допускается объявить переменную вне цикла и сразу присвоить ей начальное значение. В этом случае параметр <Начальное значение> можно оставить пустым. Пример:

```
let i; // Объявление вне цикла
for (i = 1; i <= 20; i++) {
  console.log(i);
}
// Переменная i видна вне цикла
console.log(i); // 21
// Объявление внутри цикла
for (let j = 1; j <= 20; j++) {
  console.log(j);
}
// Переменная j НЕ видна вне цикла
console.log(typeof j); // undefined
let k = 1; // Инициализация вне цикла
for ( ; k <= 20; k++) {
  console.log(k);
}
```

Если мы объявляем переменную в первом параметре цикла `for` с помощью ключевого слова `var`, то переменная будет видна и после закрывающей фигурной скобки:

```
for (var i = 0; i < 5; i++) {
  console.log(i);
}
console.log('i =', i); // i = 5
```

Цикл выполняется до тех пор, пока `<Условие>` не вернет `false`. Если это не произойдет, цикл станет бесконечным.

Логическое выражение, указанное в параметре `<Условие>`, вычисляется на каждой итерации. Рассмотрим вывод элементов массива:

```
let arr = [ 1, 2, 3 ];
for (let i = 0; i < arr.length; i++) {
  if (i == 0) {
    arr.push(4); // Добавляем новые элементы
    arr.push(5); // для доказательства
  }
  process.stdout.write(`${arr[i]} `);
} // Выведет: 1 2 3 4 5
```

В этом примере мы указываем свойство `length` в параметре `<Условие>`, а внутри цикла добавляем новые элементы в массив. В итоге получим все элементы массива, включая вновь добавленные. Поэтому, если внутри логического выражения производятся какие-либо вычисления и значение внутри цикла не изменяется, вычисление следует вынести в параметр `<Начальное значение>`. В этом случае вычисление указывается после присваивания значения переменной-счетчику через запятую. Пример:

```
let arr = [ 1, 2, 3 ];
for (let i = 0, c = arr.length; i < c; i++) {
```

```
if (i == 0) {
    arr.push(4); // Добавляем новые элементы
    arr.push(5); // для доказательства
}
process.stdout.write(`${arr[i]} `);
} // Выведет: 1 2 3
```

Выражение, указанное в параметре <Приращение>, может не только увеличивать значение переменной-счетчика, но и уменьшать его. Кроме того, значение может изменяться на любую величину. Пример:

```
// Выводим числа от 10 до 1
for (let i = 10; i > 0; i--) {
    console.log(i);
}
// Выводим четные числа от 1 до 10
for (let j = 2; j <= 10; j += 2) {
    console.log(j);
}
```

Все параметры цикла `for` являются необязательными. Хотя их можно не указывать, точки с запятой обязательно должны быть. Если все параметры отсутствуют, цикл станет бесконечным. Чтобы выйти из бесконечного цикла, следует использовать оператор `break`. Пример:

```
let i = 1; // <Начальное значение>
for ( ; ; ) { // Бесконечный цикл
    if (i <= 10) { // <Условие>
        console.log(i);
        i++; // <Приращение>
    }
    else {
        break; // Выходим из цикла
    }
}
```

Для быстрой вставки цикла `for` в редакторе VS Code предназначена следующая комбинация:

```
for =>
for (let index = 0; index < array.length; index++) {
    const element = array[index];
}
```

4.12. Цикл *while*

Выполнение инструкций в цикле `while` продолжается до тех пор, пока заданное условие истинно. Цикл имеет следующий формат:

```
<Начальное значение>
while (<Условие>) {
    <Инструкции>
    <Приращение>
}
```

Последовательность работы цикла `while`:

1. Переменной-счетчику присваивается начальное значение.
2. Проверяется условие; если оно истинно, выполняются инструкции внутри цикла, иначе выполнение цикла завершается.
3. Переменная-счетчик изменяется на величину, указанную в параметре `<Приращение>`.
4. Переход к п. 2.

Выведем все числа от 1 до 100, используя цикл `while` (листинг 4.5).

Листинг 4.5. Цикл `while`

```
let i = 1;                // <Начальное значение>
while (i < 101) {         // <Условие>
    console.log(i);       // <Инструкции>
    i++;                  // <Приращение>
}
```

ВНИМАНИЕ!

Если `<Приращение>` не указано, цикл станет бесконечным.

Для быстрой вставки цикла `while` в редакторе VS Code предусмотрена следующая комбинация:

```
while =>
while (condition) {

}
```

4.13. Цикл `do...while`

Выполнение инструкций в цикле `do...while` продолжается до тех пор, пока заданное условие истинно. В отличие от цикла `while`, условие проверяется не в начале, а в конце цикла. По этой причине инструкции внутри цикла `do...while` выполнятся как минимум один раз. Цикл имеет следующий формат:

```
<Начальное значение>
do {
    <Инструкции>
    <Приращение>
} while (<Условие>);
```

Последовательность работы цикла `do...while`:

1. Переменной-счетчику присваивается начальное значение.
2. Выполняются инструкции внутри цикла.
3. Переменная-счетчик изменяется на величину, указанную в параметре `<Приращение>`.
4. Проверяется условие; если оно истинно, происходит переход к п. 2, а если нет — выполнение цикла завершается.

Выведем все числа от 1 до 100, используя цикл `do...while` (листинг 4.6).

Листинг 4.6. Цикл `do...while`

```
let i = 1;                // <Начальное значение>
do {
  console.log(i);         // <Инструкции>
  i++;                    // <Приращение>
} while (i < 101);         // <Условие>
```

ВНИМАНИЕ!

Если `<Приращение>` не указано, цикл станет бесконечным.

Для быстрой вставки цикла `do...while` в редакторе VS Code предусмотрена следующая комбинация:

```
dowhile =>
do {

} while (condition);
```

4.14. Цикл `for...in`

Цикл `for...in` служит для перебора перечислимых свойств объектов и ключей ассоциативных массивов. Он имеет следующий формат:

```
for (<Переменная> in <Объект>) {
  <Тело цикла>
}
```

Цикл `for...in` на каждой итерации присваивает `<Переменной>` имя свойства объекта или ключ ассоциативного массива. Пример перебора свойств объекта приведен в листинге 4.7.

Листинг 4.7. Цикл `for...in`

```
let obj = { a: 10, b: 20 };
for (let prop in obj) {
  process.stdout.write(`${prop} `);
} // a b
```


При переборе ассоциативного массива с помощью ключа можно получить значение соответствующего элемента:

```
let arr = new Object();
arr['Один'] = 1;
arr['Два'] = 2;
arr['Три'] = 3;
for (let key in arr) {
  // Переменной key на каждой итерации присваивается
  // ключ текущего элемента ассоциативного массива
  console.log(`${key} = ${arr[key]}`);
}
```

В итоге мы получим следующий результат:

```
Один = 1
Два = 2
Три = 3
```

Следует учитывать, что порядок обхода является произвольным, поэтому перебирать обычные массивы, где порядок следования элементов важен, с помощью этого цикла не стоит.

Цикл `for...in` имеет доступ не только к свойствам текущего объекта, но и к унаследованным свойствам в цепочке прототипов. Если эти унаследованные свойства не нужны, то следует отфильтровать их с помощью метода `hasOwnProperty()`:

```
let objA = { a: 10 };
let objB = { b: 55 };
Object.setPrototypeOf(objB, objA);
console.log(objB.a); // 10
console.log(objB.b); // 55
for (let prop in objB) {
  if ( objB.hasOwnProperty(prop) ) {
    console.log(`YES: ${prop}`);
  }
  else {
    console.log(`NO: ${prop}`);
  }
}
// YES: b
// NO: a
```

ОБРАТИТЕ ВНИМАНИЕ

Перебрать можно только перечислимые свойства, т. е. свойства, для которых метод `propertyIsEnumerable()` из класса `Object` вернул значение `true`. Для свойств встроенных объектов метод возвращает `false`, поэтому перебрать свойства этих объектов с помощью цикла `for...in` нельзя.

Для быстрой вставки цикла `for...in` в редакторе VS Code предусмотрена следующая комбинация:

```
forin =>
for (const key in object) {
  if (object.hasOwnProperty(key)) {
    const element = object[key];

  }
}
```

4.15. Цикл *for...of*

Цикл `for...of` используется для перебора объектов, поддерживающих итерации, например массивов и строк. Он имеет следующий формат:

```
for (<Переменная> of <Объект>) {
  <Тело цикла>
}
```

Пример перебора элементов массива приведен в листинге 4.8.

Листинг 4.8. Цикл `for...of`

```
let arr = [1, 2, 3];
for (let value of arr) {
  process.stdout.write(`${value} `);
} // 1 2 3
```

Выполним перебор букв внутри строки:

```
let str = 'abc';
for (let letter of str) {
  process.stdout.write(`${letter} `);
} // a b c
```

Для быстрой вставки цикла `for...of` в редакторе VS Code предусмотрена следующая комбинация:

```
forof =>
for (const iterator of object) {

}
```

4.16. Оператор *continue*: переход на следующую итерацию цикла

Оператор `continue` позволяет перейти на следующую итерацию цикла до завершения выполнения всех инструкций внутри цикла. Этот оператор можно применять в любых циклах.

Выведем все числа от 1 до 100, кроме чисел от 5 до 10 включительно (листинг 4.9).

Листинг 4.9. Использование оператора `continue`

```
for (let i = 1; i < 101; i++) {  
  if (i > 4 && i < 11) continue;  
  console.log(i);  
}
```

4.17. Оператор *break*: прерывание цикла

Оператор `break` позволяет прервать выполнение цикла досрочно.

Выведем все числа от 1 до 100 еще одним способом (листинг 4.10).

Листинг 4.10. Прерывание цикла

```
let i = 1;  
while (true) {  
  if (i > 100) break;  
  console.log(i);  
  i++;  
}
```

Здесь мы в условии указали значение `true`. В этом случае инструкции внутри цикла будут выполняться бесконечно. Однако использование оператора `break` прерывает его выполнение, как только будет напечатано 100 строк.

ОБРАТИТЕ ВНИМАНИЕ

Оператор `break` прерывает выполнение цикла, а не программы, т. е. далее будет выполнена инструкция, следующая сразу за циклом.



ГЛАВА 5

Числа

В языке JavaScript числа относятся к одному из следующих типов данных:

- ❑ **number** — числа, целые или с плавающей точкой (вещественные). Хранятся в формате вещественных чисел двойной точности. Пример:

```
console.log(typeof 1);           // number
console.log(typeof 2.0);         // number
```

- ❑ **bigint** — целые числа больших размеров:

```
console.log(typeof 1n);          // bigint
```

5.1. Указание значений

Целочисленное значение может быть задано в десятичной, двоичной, восьмеричной или шестнадцатеричной системе счисления. Двоичные числа начинаются с комбинации символов `0b` (или `0B`) и могут содержать цифры 0 и 1. Восьмеричные числа начинаются с нуля или комбинации символов `0o` и содержат цифры от 0 до 7. Если встречаются другие цифры, то значение интерпретируется как десятичное. Шестнадцатеричные числа начинаются с комбинации символов `0x` (или `0X`) и могут содержать цифры от 0 до 9 и буквы от `A` до `F` (регистр букв не имеет значения). Двоичные, восьмеричные и шестнадцатеричные значения преобразуются в десятичное значение. Пример:

```
// Двоичное значение
console.log(0b01110111);          // 119
// Восьмеричное значение
console.log(0o167);               // 119
// Шестнадцатеричное значение
console.log(0x77);                 // 119
console.log(0xFF);                 // 255
// Десятичное значение
console.log(119);                  // 119
```

Для отделения групп цифр друг от друга в числах можно использовать символы подчеркивания:

```
console.log(1_000_000);           // 1000000
console.log(0xff_ed);             // 65517
```

Восьмеричное значение может начинаться с нуля, но в строгом режиме данный способ приведет к исключению:

```
'use strict';
console.log(0o167);               // OK. 119
// console.log(0167);
// SyntaxError: Octal literals are not allowed in strict mode.
```

Вещественное число может содержать точку и (или) экспоненту, начинающуюся с буквы E (регистр не имеет значения):

```
console.log(20.0);               // 20
console.log(12.1e20);            // 1.21e+21
console.log(.123);               // 0.123
console.log(47.E-10);            // 4.7e-9
```

При выполнении операций над вещественными числами следует учитывать ограничения точности вычислений. Например, результат следующей инструкции может показаться странным:

```
console.log(0.3 - 0.1 - 0.1 - 0.1);
// -2.7755575615628914e-17
```

Ожидаемым был бы результат 0.0, но, как видно из примера, мы получили совсем другой результат (-2.7755575615628914e-17). Он очень близок к нулю, но не равен ему. Учитывайте это при указании вещественных чисел в качестве значения счетчика внутри цикла. Попытка проверить значение на равенство может привести к бесконечному циклу.

В логическом контексте число 0 и значение NaN трактуются как false, тогда как любое другое число — как true:

```
console.log( !!0 );               // false
console.log( !!0.0 );            // false
console.log( !!NaN );            // false
console.log( !!1 );              // true
console.log( !!-1 );             // true
```

Если выполнить приведение логических значений к типу number, то false будет приведено к нулю, а true — к единице:

```
console.log( false + 0 );        // 0
console.log( true + 0 );         // 1
```

5.2. Класс *Number*

Класс `Number` является классом-"оберткой" над элементарным типом `number`. Экземпляр класса создается по следующей схеме:

```
<Экземпляр класса> = new Number(<Начальное значение>);
```

Пример:

```
let x = new Number(10);
console.log(x.toString()); // 10
console.log(typeof x);     // object
```

Однако класс `Number` применяют редко, поскольку все его свойства и методы доступны и при использовании элементарного типа. Преобразование элементарного типа в объектный выполняется автоматически. Пример:

```
let x = 255;
console.log(x.toString(16)); // ff
console.log(typeof x);      // number
```

Если в операции участвуют число и экземпляр класса `Number`, то экземпляр класса будет автоматически преобразован в число:

```
let x = new Number(10);
let y = 255;
console.log(x + y);      // 265
```

Для преобразования значения в число применяется следующий формат:

```
<Переменная> = Number(<Значение>);
```

Пример:

```
let x = Number('10');
console.log(x);           // 10
console.log(typeof x);    // number
```

Свойства класса `Number` можно использовать без создания экземпляра класса:

❑ `MAX_VALUE` — максимально допустимое в JavaScript число:

```
console.log(Number.MAX_VALUE); // 1.7976931348623157e+308
```

❑ `MIN_VALUE` — минимально допустимое в JavaScript число:

```
console.log(Number.MIN_VALUE); // 5e-324
```

Обратите внимание на эту особенность языка:

```
console.log(Number.MIN_VALUE < 0); // false, а не true !
```

❑ `MIN_SAFE_INTEGER` — минимальное целочисленное значение:

```
console.log(Number.MIN_SAFE_INTEGER); // -9007199254740991
```

❑ `MAX_SAFE_INTEGER` — максимальное целочисленное значение:

```
console.log(Number.MAX_SAFE_INTEGER); // 9007199254740991
```

❑ `NaN` — значение NaN:

```
console.log(Number.NaN); // NaN
```

❑ `NEGATIVE_INFINITY` — значение $-\infty$:

```
console.log(Number.NEGATIVE_INFINITY); // -Infinity
```

- ❑ `POSITIVE_INFINITY` — значение ∞ :

```
console.log(Number.POSITIVE_INFINITY); // Infinity
```

- ❑ `EPSILON` — наименьший интервал между двумя представимыми числами:

```
console.log(Number.EPSILON); // 2.220446049250313e-16
```

Методы:

- ❑ `valueOf()` — возвращает числовое значение экземпляра класса:

```
let x = new Number(15);
let y = x.valueOf(); // 15
console.log(typeof y); // number
```

- ❑ `toString([<Основание>])` — возвращает строковое представление числа в системе счисления с заданным в параметре основанием. Если основание не указано, выданное число будет представлено в десятичной системе счисления. Пример:

```
let x = 15;
let str = x.toString(); // "15"
console.log(typeof str); // string
x = 100;
console.log(x.toString(2)); // 1100100
x = 119;
console.log(x.toString(8)); // 167
x = 255;
console.log(x.toString(16)); // ff
```

- ❑ `toFixed([<Количество>])` — возвращает строковое представление числа, округленное до указанного количества цифр после точки. Если параметр не указан, значение округляется до целого числа. Пример:

```
let x = 1234.6789;
console.log(x.toFixed()); // 1235
console.log(x.toFixed(1)); // 1234.7
console.log(x.toFixed(2)); // 1234.68
```

- ❑ `toExponential([<Количество>])` — возвращает строковое представление числа в экспоненциальной форме, округленное до указанного количества цифр после точки. Если параметр не указан, округление не выполняется. Пример:

```
let x = 0.06789;
console.log(x.toExponential()); // 6.789e-2
console.log(x.toExponential(1)); // 6.8e-2
console.log(x.toExponential(2)); // 6.79e-2
```

- ❑ `toPrecision([<Количество>])` — возвращает строковое представление числа, округленное до указанного количества значащих цифр. Если параметр не указан, метод аналогичен методу `toString()`. Пример:

```
let x = 1234.6789;
console.log(x.toPrecision()); // 1234.6789
console.log(x.toPrecision(5)); // 1234.7
```

```
console.log(x.toPrecision(4));    // 1235
console.log(x.toPrecision(3));    // 1.23e+3
console.log(x.toPrecision(2));    // 1.2e+3
console.log(x.toPrecision(1));    // 1e+3
```

- ❑ `toLocaleString()` — возвращает строковое представление числа, оформленное в соответствии с локалью операционной системы:

```
let x = 1027324.564;
console.log(x.toLocaleString()); // 1 027 324,564
```

- ❑ `isInteger(<Значение>)` — возвращает `true`, если указанное значение является целым числом или вещественным числом, которое может быть преобразовано в целое без потерь:

```
console.log(Number.isInteger(10)); // true
console.log(Number.isInteger(10.0)); // true
console.log(Number.isInteger(10.2)); // false
console.log(Number.isInteger('10')); // false
```

- ❑ `isSafeInteger(<Значение>)` — возвращает `true`, если указанное значение является "безопасным" целым числом (находящимся в диапазоне от $-2^{53}-1$ до $2^{53}-1$):

```
console.log(Number.isSafeInteger(10)); // true
console.log(Number.isSafeInteger(Number.MAX_SAFE_INTEGER + 1)); // false
```

- ❑ `parseInt(<Строка>[, <Основание>])` — преобразует заданную строку в целое число и возвращает его (аналог глобальной функции `parseInt()`). Строка считается заданной в системе счисления, основание которой указано вторым параметром. Если основание не указано, система счисления выбирается автоматически и зависит от префиксов в строке (например, если строка начинается с `0x`, будет использоваться основание 16). Если строка не может быть преобразована в число, возвращается значение `NaN`. Пример:

```
console.log( Number.parseInt('10') ); // 10
console.log( Number.parseInt('0xFF') ); // 255
console.log( Number.parseInt('0xFF', 16) ); // 255
console.log( Number.parseInt('0167', 8) ); // 119
console.log( Number.parseInt('1110111', 2) ); // 119
console.log( Number.parseInt('строка', 2) ); // NaN
```

- ❑ `parseFloat(<Строка>)` — преобразует заданную строку в число с плавающей точкой и возвращает его (аналог глобальной функции `parseFloat()`):

```
console.log( Number.parseFloat('5.2') ); // 5.2
console.log( Number.parseFloat('5e-2') ); // 0.05
console.log( Number.parseFloat('строка') ); // NaN
```

- ❑ `isFinite(<Значение>)` — возвращает `true`, если заданное значение не равно ∞ , $-\infty$ или `NaN`, и `false` — в противном случае (аналог глобальной функции `isFinite()`). В отличие от глобальной функции `isFinite()`, автоматически не преобразует указанное значение в число. Пример:


```
console.log( Number.isFinite('0') );           // false
console.log( isFinite('0') );                 // true
```

- ❑ `isNaN(<Значение>)` — возвращает `true`, если заданное значение равно `NaN`, и `false` — в противном случае (аналог глобальной функции `isNaN()`):

```
let x = 10.0 / 0, y = 0.0 / 0, z = 5;
console.log( x + ' ' + y + ' ' + z ); // Infinity NaN 5
console.log( Number.isNaN(x) );       // false
console.log( Number.isNaN(y) );       // true
console.log( Number.isNaN(z) );       // false

console.log( Number.isFinite(x) );    // false
console.log( Number.isFinite(y) );    // false
console.log( Number.isFinite(z) );    // true
```

В отличие от глобальной функции `isNaN()`, метод `isNaN()` автоматически не преобразует указанное значение в число:

```
console.log( Number.isNaN('str') ); // false
console.log( isNaN('str') );         // true
```

5.3. Математические константы

В классе `Math` определены следующие стандартные константы:

- ❑ `PI` — число π :

```
console.log(Math.PI); // 3.141592653589793
```

- ❑ `E` — число e (основание натурального логарифма):

```
console.log(Math.E); // 2.718281828459045
```

- ❑ `LN2` — $\ln 2$:

```
console.log(Math.LN2); // 0.6931471805599453
```

- ❑ `LN10` — $\ln 10$:

```
console.log(Math.LN10); // 2.302585092994046
```

- ❑ `LOG2E` — $\log_2 e$:

```
console.log(Math.LOG2E); // 1.4426950408889634
```

- ❑ `LOG10E` — $\lg e$:

```
console.log(Math.LOG10E); // 0.4342944819032518
```

- ❑ `SQRT1_2` — квадратный корень из 0,5:

```
console.log(Math.SQRT1_2); // 0.7071067811865476
```

- ❑ `SQRT2` — квадратный корень из 2:

```
console.log(Math.SQRT2); // 1.4142135623730951
```

5.4. Основные методы для работы с числами

В классе `Math` определены следующие основные методы для работы с числами:

- ❑ `abs(<Значение>)` — абсолютное значение:

```
console.log( Math.abs(-5) );    // 5
```

- ❑ `exp(<Значение>)` — экспонента:

```
console.log( Math.exp(-1) );    // 0.36787944117144233
```

- ❑ `expm1(<Значение>)` — экспонента, из которой вычли единицу:

```
console.log( Math.expm1(-1) );  // -0.6321205588285577
```

- ❑ `log(<Значение>)` — натуральный логарифм:

```
console.log( Math.log(10) );    // 2.302585092994046
```

- ❑ `log1p(<Значение>)` — натуральный логарифм от заданного значения, к которому прибавлена единица:

```
console.log( Math.log1p(10) );  // 2.3978952727983707
```

- ❑ `log2(<Значение>)` — двоичный логарифм:

```
console.log( Math.log2(2) );    // 1
```

- ❑ `log10(<Значение>)` — десятичный логарифм:

```
console.log( Math.log10(1) );   // 0
```

- ❑ `pow(<Число>, <Степень>)` — возведение <Числа> в <Степень> (аналог оператора `**`):

```
let x = 5;  
console.log( Math.pow(x, 2) );  // 25 (5 в квадрате)
```

- ❑ `sqrt(<Значение>)` — квадратный корень:

```
let x = 25;  
console.log( Math.sqrt(x) );    // 5 (квадратный корень из 25)
```

- ❑ `cbrt(<Значение>)` — кубический корень:

```
let x = 125;  
console.log( Math.cbrt(x) );    // 5 (кубический корень из 125)
```

- ❑ `max(<Список чисел через запятую>)` — максимальное значение из списка:

```
console.log( Math.max(3, 10, 6) ); // 10  
let arr = [3, 10, 6];  
console.log( Math.max(...arr) );   // 10
```

- ❑ `min(<Список чисел через запятую>)` — минимальное значение из списка:

```
console.log( Math.min(3, 10, 6) ); // 3  
let arr = [3, 10, 6];  
console.log( Math.min(...arr) );   // 3
```

- ❑ `hypot(<Список чисел через запятую>)` — квадратный корень из суммы квадратов указанных чисел:

```
console.log( Math.hypot(3, 4) );    // 5
console.log( Math.hypot(3, 4, 5) ); // 7.0710678118654755
```

- ❑ `sign(<Значение>)` — возвращает `-1`, если заданное значение отрицательное, `0` — если оно равно `0`, `1` — если значение положительное, и `NaN`, если задано значение `NaN`:

```
console.log( Math.sign(100) );    // 1
console.log( Math.sign(0) );      // 0
console.log( Math.sign(-100) );   // -1
```

5.5. Округление чисел

Для округления чисел предназначены следующие методы из класса `Math`:

- ❑ `ceil()` — возвращает значение, округленное до ближайшего большего целого:

```
console.log( Math.ceil(1.49) );    // 2
console.log( Math.ceil(1.50) );    // 2
console.log( Math.ceil(1.51) );    // 2
```

- ❑ `floor()` — возвращает значение, округленное до ближайшего меньшего целого:

```
console.log( Math.floor(1.49) );    // 1
console.log( Math.floor(1.50) );    // 1
console.log( Math.floor(1.51) );    // 1
```

- ❑ `round()` — возвращает число, округленное до ближайшего меньшего целого, для чисел с дробной частью меньше `0,5`, или значение, округленное до ближайшего большего целого, для чисел с дробной частью больше или равной `0,5`:

```
console.log( Math.round(1.49) );    // 1
console.log( Math.round(1.50) );    // 2
console.log( Math.round(1.51) );    // 2
```

- ❑ `trunc()` — отбрасывает дробную часть и возвращает целое число:

```
console.log( Math.trunc(1.5) );      // 1
console.log( Math.trunc(-1.5) );     // -1
console.log( Math.ceil(1.5) );       // 2
console.log( Math.ceil(-1.5) );      // -1
console.log( Math.floor(1.5) );      // 1
console.log( Math.floor(-1.5) );     // -2
```

5.6. Тригонометрические функции

В классе `Math` определены следующие основные тригонометрические методы:

- ❑ `sin()`, `cos()`, `tan()` — стандартные тригонометрические функции (синус, косинус, тангенс). Угол задается в радианах. Пример:

```
let n = Math.PI / 180;
console.log( Math.sin(90 * n) );    // 1
```

```
console.log( Math.cos(10 * n) );    // 0.984807753012208
console.log( Math.tan(45 * n) );    // 0.9999999999999999
```

- ❑ `asin()`, `acos()`, `atan()` — обратные тригонометрические функции (арксинус, арккосинус, арктангенс). Значение возвращается в радианах. Пример:

```
console.log( Math.asin(1) );        // 1.5707963267948966
console.log( Math.acos(-1) );       // 3.141592653589793
console.log( Math.atan(1) );        // 0.7853981633974483
```

- ❑ `sinh()`, `cosh()`, `tanh()` — гиперболические тригонометрические функции (синус, косинус, тангенс). Угол задается в радианах. Пример:

```
let n = Math.PI / 180;
console.log( Math.sinh(90 * n) );   // 2.3012989023072947
console.log( Math.cosh(10 * n) );   // 1.015269573507533
console.log( Math.tanh(45 * n) );   // 0.6557942026326724
```

- ❑ `asinh()`, `acosh()`, `atanh()` — обратные гиперболические тригонометрические функции (арксинус, арккосинус, арктангенс). Значение возвращается в радианах. Пример:

```
console.log( Math.asinh(1) );       // 0.881373587019543
console.log( Math.acosh(1) );       // 0
console.log( Math.atanh(0.5) );     // 0.5493061443340548
```

5.7. Преобразование строки в число

Числа, вводимые пользователем в конце команды, представлены в виде строки. Чтобы в дальнейшем использовать такие числа, необходимо преобразовать их в числовой тип. Для этого предназначены следующие функции:

- ❑ `parseInt(<Строка>[, <Основание>])` — преобразует указанную строку в целое число. Строка считается заданной в системе счисления с основанием, указанным вторым параметром. Если основание не задано, система счисления выбирается автоматически и зависит от префиксов в строке (например, если строка начинается с `0x`, будет использоваться основание 16). Если строка не может быть преобразована в число, возвращается значение `NaN`. Пример:

```
console.log( parseInt('10') );      // 10
console.log( parseInt('0xFF') );    // 255
console.log( parseInt('0xFF', 16) ); // 255
console.log( parseInt('0167', 8) ); // 119
console.log( parseInt('0o167', 8) ); // 0, а не 119 !
console.log( parseInt('1110111', 2) ); // 119
console.log( parseInt('строка', 2) ); // NaN
```

Можно также воспользоваться статическим методом `parseInt()` из класса `Number`:

```
console.log( Number.parseInt('10') ); // 10
console.log( Number.parseInt('0xFF') ); // 255
```

```

console.log( Number.parseInt('0xFF', 16) ); // 255
console.log( Number.parseInt('0167', 8) ); // 119
console.log( Number.parseInt('0o167', 8) ); // 0, а не 119 !
console.log( Number.parseInt('1110111', 2) ); // 119
console.log( Number.parseInt('строка', 2) ); // NaN

```

□ `parseFloat(<Строка>)` — преобразует заданную строку в число с плавающей точкой:

```

console.log( parseFloat('5.2') ); // 5.2
console.log( parseFloat('5e-2') ); // 0.05
console.log( parseFloat('строка') ); // NaN

```

Можно также воспользоваться статическим методом `parseFloat()` из класса `Number`:

```

console.log( Number.parseFloat('5.2') ); // 5.2
console.log( Number.parseFloat('5e-2') ); // 0.05
console.log( Number.parseFloat('строка') ); // NaN

```

Доступны также следующие способы преобразования:

```

console.log( Number('10') ); // 10
console.log( Number('0xFF') ); // 255
console.log( Number('0167') ); // 167, а не 119 !
console.log( Number('0o167') ); // 119, а не 0 как в parseInt()
console.log( Number('5.2') ); // 5.2
console.log( Number('5e-2') ); // 0.05
console.log( Number('строка') ); // NaN
let a = '10';
let n = +a;
console.log( n ); // 10
console.log( typeof n ); // number
console.log( +'5.2' ); // 5.2

```

5.8. Преобразование числа в строку

Чтобы преобразовать число в строку, достаточно воспользоваться операцией конкатенации строк или выполнить подстановку значения:

```

let x = 10, y = 1.2;
console.log(x + ' ' + y); // 10 1.2
console.log(`${x} ${y}`); // 10 1.2

```

Можно также воспользоваться методом `toString(<Основание>)`:

```

let x = 15;
let str = x.toString(); // "15"
console.log(typeof str); // string
x = 100;
console.log(x.toString(2)); // 1100100
x = 119;

```

```
console.log(x.toString(8)); // 167
x = 255;
console.log(x.toString(16)); // ff
```

А также методами `toFixed()`, `toPrecision()` и `toExponential()` из класса `Number` (см. разд. 5.2):

```
let x = 0.06789;
console.log(x.toFixed(2)); // 0.07
console.log(x.toPrecision(2)); // 0.068
console.log(x.toExponential(2)); // 6.79e-2
```

Метод `toLocaleString(<Локаль>[, <Настройки>])` возвращает строковое представление числа в зависимости от локали, указанной в первом параметре. Если локаль не задана, используется локаль операционной системы. Пример:

```
let x = 1027324.564;
console.log(x.toLocaleString()); // 1 027 324,564
console.log(x.toLocaleString('ru-RU')); // 1 027 324,564
console.log(x.toLocaleString('en-US')); // 1,027,324.564
console.log(x.toLocaleString('de-DE')); // 1.027.324,564
```

Во втором параметре указывается объект с дополнительными настройками:

- ❑ `style` — стиль форматирования: `'decimal'` (просто число; значение по умолчанию), `'currency'` (валюта) или `'percent'` (проценты);
- ❑ `currency` — задает код валюты для стиля `'currency'`. Список кодов валют можно найти на странице: <https://index.minfin.com.ua/reference/currency/code/>;
- ❑ `currencyDisplay` — задает способ отображения валюты для стиля `'currency'`: `'symbol'` (символ валюты, например `'`; значение по умолчанию), `'code'` (код валюты, например `'RUB'`) или `'name'` (название валюты, например `"российский рубль"`):

```
let x = 1027324.5647;
console.log( x.toLocaleString('ru-RU',
    { style: 'currency', currency: 'RUB' } ) ); // '1 027 324,56 '
console.log( x.toLocaleString('ru-RU',
    { style: 'currency', currency: 'RUB',
      currencyDisplay: 'name' } ) );
// '1 027 324,56 российского рубля'
```

- ❑ `minimumIntegerDigits` — минимальное количество цифр в целой части числа (значение по умолчанию: 1);
- ❑ `minimumFractionDigits` — минимальное количество цифр в дробной части числа (значение по умолчанию: 0);
- ❑ `maximumFractionDigits` — максимальное количество цифр в дробной части числа:

```
let x = 1027324.5647;
console.log( x.toLocaleString('ru-RU',
    { minimumIntegerDigits: 1,
```

```
minimumFractionDigits: 0,
maximumFractionDigits: 2 }) ); // 1 027 324,56
```

- ❑ `minimumSignificantDigits` — минимальное количество значащих цифр числа в диапазоне от 1 до 21 (по умолчанию: 1);
- ❑ `maximumSignificantDigits` — максимальное количество значащих цифр числа в диапазоне от 1 до 21;

```
console.log( x.toLocaleString('ru-RU',
    { minimumSignificantDigits: 15,
      maximumSignificantDigits: 20 }) ); // 1 027 324,56470000
```

- ❑ `useGrouping` — указывает, выводить ли разделители тысячных групп. Можно указать значения `true` (разделители выводятся; по умолчанию) или `false` (разделители не выводятся). Пример:

```
let x = 1027324.5647;
console.log( x.toLocaleString('ru-RU',
    { useGrouping: false,
      maximumFractionDigits: 2 }) ); // 1027324,56
```

Вместо метода `toLocaleString()` можно воспользоваться классом `Intl.NumberFormat`:

```
<Экземпляр класса> = new Intl.NumberFormat([<Локаль>[, <Настройки>]]);
```

Применить настройки и получить отформатированную строку позволяет метод `format(<Число>)`:

```
let x = 1027324.5647;
let obj = new Intl.NumberFormat('ru-RU');
console.log(obj.format(x)); // 1 027 324,565
obj = new Intl.NumberFormat('ru-RU', { useGrouping: false,
                                         maximumFractionDigits: 2 });
console.log(obj.format(x)); // 1027324,56
```

Метод `resolvedOptions()` возвращает объект со всеми настройками:

```
let obj = new Intl.NumberFormat('ru-RU');
console.log(obj.resolvedOptions());
// {
//   locale: 'ru-RU',
//   numberingSystem: 'latn',
//   style: 'decimal',
//   minimumIntegerDigits: 1,
//   minimumFractionDigits: 0,
//   maximumFractionDigits: 2,
//   useGrouping: false,
//   notation: 'standard',
//   signDisplay: 'auto'
// }
```

5.9. Генерация псевдослучайных чисел

Для генерации псевдослучайных чисел предназначен метод `random()` из класса `Math`. Он возвращает псевдослучайное число от 0 до 1. Пример:

```
console.log( Math.random() ); // Например, 0.7110987103682411
```

Чтобы получить случайное целое число от 0 до 9, нужно возвращенное методом `random()` значение умножить на 9,9999, а затем округлить число до ближайшего меньшего целого при помощи метода `floor()`:

```
console.log( Math.floor(Math.random() * 9.9999) ); // Например, 2
```

Попробуйте запустить программу несколько раз. Число будет меняться случайным образом в пределах от 0 до 9 включительно.

Если нужно, чтобы число генерировалось в диапазоне от `min` до `max` включительно, можно воспользоваться следующим кодом:

```
let min = 5, max = 10;
let n = Math.floor(Math.random() * (max - min + 1) + min)
console.log( n ); // Например, 7
```

5.10. Бесконечность и значение NaN

Деление числа на 0 приведет к значению плюс или минус `Infinity` (бесконечность), а деление числа 0.0 на 0 — к значению `NaN` (нет числа):

```
console.log( 10 / 0 ); // Infinity
console.log( -10 / 0 ); // -Infinity
console.log( 0 / 0 ); // NaN
```

Присвоить эти значения можно также с помощью глобальных свойств `Infinity` и `NaN` или свойств `POSITIVE_INFINITY`, `NEGATIVE_INFINITY` и `NaN` из класса `Number`:

```
console.log( Infinity ); // Infinity
console.log( -Infinity ); // -Infinity
console.log( NaN ); // NaN
console.log(Number.POSITIVE_INFINITY); // Infinity
console.log(Number.NEGATIVE_INFINITY); // -Infinity
console.log(Number.NaN); // NaN
```

Любая операция со значением `Infinity` с числом даст в результате `Infinity`:

```
console.log( Infinity - 5 ); // Infinity
```

Любая операция со значением `NaN` даст в результате `NaN`:

```
console.log( NaN + 5 ); // NaN
```

При сравнении значение `NaN` не равно даже самому себе:

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


Для проверки соответствия этим значениям следует воспользоваться следующими глобальными функциями:

- `isFinite(<Значение>)` — возвращает `true`, если заданное значение не равно ∞ , $-\infty$ или значению `NaN`, и `false` — в противном случае;
- `isNaN(<Значение>)` — возвращает `true`, если заданное значение равно `NaN`, и `false` — в противном случае:

```
let x = 10.0 / 0, y = 0.0 / 0, z = 5;
console.log( x + ' ' + y + ' ' + z ); // Infinity NaN 5
console.log( Number.isNaN(x) );      // false
console.log( Number.isNaN(y) );      // true
console.log( Number.isNaN(z) );      // false

console.log( Number.isFinite(x) );    // false
console.log( Number.isFinite(y) );    // false
console.log( Number.isFinite(z) );    // true
```

Выполнить сравнение со значением `NaN` позволяет также статический метод `is(<Значение 1>, <Значение 2>)` из класса `Object`:

```
console.log( Object.is(x, NaN) );     // false
console.log( Object.is(y, NaN) );     // true
console.log( Object.is(z, NaN) );     // false
```

Можно также воспользоваться статическими методами `isFinite()` и `isNaN()` из класса `Number` (см. *разд. 5.2*):

```
let x = 10.0 / 0, y = 0.0 / 0, z = 5;
console.log( Number.isNaN(y) );       // true
console.log( Number.isFinite(x) );     // false
console.log( Number.isFinite(z) );     // true
```

Обратите внимание: в отличие от глобальных функций `isFinite()` и `isNaN()`, методы `isFinite()` и `isNaN()` из класса `Number` автоматически не преобразуют указанное значение в число:

```
console.log( Number.isFinite('0') );  // false
console.log( isFinite('0') );          // true
console.log( Number.isNaN('str') );    // false
console.log( isNaN('str') );           // true
```

5.11. Тип данных *bigint*

Тип данных `bigint` позволяет работать с целыми числами больших размеров. Чтобы создать число этого типа, нужно добавить букву `n` после целочисленного литерала. Пример:

```
let n = 10n;
console.log(n);           // 10n
console.log(typeof n);    // bigint
```

Можно также воспользоваться функцией `BigInt(<Значение>)`, которая преобразует указанное значение в число типа `bigint`:

```
let n = BigInt('10');
console.log(n);           // 10n
console.log(typeof n);    // bigint
```

Если преобразование невозможно, будет сгенерировано исключение:

```
let n = BigInt('str');
// SyntaxError: Cannot convert str to a BigInt
```

Над двумя числами типа `bigint` можно выполнять математические (кроме унарного оператора `+`) и побитовые (кроме оператора `>>>`) операции. При этом деление всегда возвращает целочисленное значение типа `bigint`, даже в том случае, если деление выполнено с остатком. Пример:

```
let n = BigInt(Number.MAX_SAFE_INTEGER);
console.log(n + 1n); // 9007199254740992n
console.log(n - 1n); // 9007199254740990n
console.log(n * 2n); // 18014398509481982n
n = 10n;
console.log(n / 3n); // 3n
console.log(n % 4n); // 2n
console.log(n ** 2n); // 100n
console.log(++n);     // 11n
console.log(--n);     // 10n
console.log(-n);      // -10n
console.log(~n);      // -11n
let x = 100n, y = 75n, z;
z = x & y;
console.log( z.toString(2) ); // 1000000
z = x | y;
console.log( z.toString(2) ); // 1101111
y = 250n;
z = x ^ y;
console.log( z.toString(2) ); // 10011110
x = x << 1n;
console.log( x.toString(2) ); // 11001000
x = x >> 1n;
console.log( x.toString(2) ); // 1100100
```

Если в операции участвуют типы `number` и `bigint`, будет сгенерировано исключение, поэтому не забудьте выполнить преобразование типов:

```
let x = 10n, y = 2, z;
z = x + y;
// TypeError: Cannot mix BigInt and other types,
// use explicit conversions
```

Исключением являются операции сравнения и логические операции:

```
let x = 10n, y = 2, z = 10;
console.log(x == z); // true
console.log(x === z); // false
console.log(x < y); // false
console.log(x > y); // true
console.log(x <= z); // true
console.log(x >= z); // true
console.log(!!x); // true
```

Чтобы преобразовать число типа `bigint` в тип `number`, следует воспользоваться функцией `Number()`, но при этом следует учитывать возможную потерю точности:

```
let x = 10n, y;
y = Number(x);
console.log(y); // 10
console.log(typeof y); // number
```

Чтобы из числа типа `bigint` получить строку, следует воспользоваться методом `toString(<Основание>)`:

```
let x = 15n;
let str = x.toString(); // "15"
console.log(typeof str); // string
x = 100n;
console.log(x.toString(2)); // 1100100
x = 119n;
console.log(x.toString(8)); // 167
x = 255n;
console.log(x.toString(16)); // ff
```

Для получения строки в зависимости от настроек локали применяется метод `toLocaleString(<Локаль>[, <Настройки>])`:

```
let x = BigInt(Number.MAX_SAFE_INTEGER);
console.log(x.toLocaleString('en-US'));
// 9,007,199,254,740,991
```



ГЛАВА 6

Массивы и множества

Массив — это пронумерованный набор значений. Отдельное значение в массиве называется *элементом*, а его порядковый номер в массиве — *индексом*. Нумерация элементов массива начинается с 0, а не с 1, — это следует помнить. Общее количество элементов в массиве называется *размером* массива. Массивы, ключами которых являются числа, часто называют *списками*, а массивы, ключами которых являются строки, — *ассоциативными массивами*.

6.1. Инициализация массива

Массив инициализируют следующими способами:

- указанием значений через запятую в квадратных скобках:

```
let arr = [1, 2, 3, 4];  
console.log( arr );           // [ 1, 2, 3, 4 ]  
console.log( typeof arr );    // object
```

Оператор **SPREAD** (три точки) позволяет указывать объекты, поддерживающие итерации, в качестве элементов при создании нового массива. Три точки следует поставить перед именем существующего объекта — и все элементы этого объекта будут вставлены в новый массив. Пример:

```
let arr1 = [1, 2];  
let arr2 = [4, 5];  
let arr3 = [...arr1, 3, ...arr2, 6];  
console.log( arr3 ); // [ 1, 2, 3, 4, 5, 6 ]
```

- с помощью класса `Array`. Класс предоставляет доступ к множеству методов для обработки массивов. Экземпляр класса можно создать следующими способами:

```
<Экземпляр класса> = new Array([<Количество элементов массива>]);  
<Экземпляр класса> = new Array(<Элементы массива через запятую>);
```

Если в круглых скобках нет никаких параметров, то создается массив нулевой длины, т. е. массив, не содержащий элементов:

```
let arr = new Array();
console.log( arr );           // []
console.log( typeof arr );    // object
```

Если указано одно целое число, оно задает количество элементов массива, при этом все элементы созданного массива будут хранить значения `undefined`:

```
let arr = new Array(3);
console.log( arr );           // [ undefined, undefined, undefined ]
```

Если указано несколько значений через запятую или единственное заданное значение не является числом, то значения заносятся в создаваемый массив:

```
let arr = new Array(1, 2, 3, 4);
console.log( arr );           // [ 1, 2, 3, 4 ]
```

□ с помощью статического метода `of(<Значения через запятую>)` из класса `Array`:

```
let arr = Array.of(1, 2, 3, 4);
console.log(arr);             // [ 1, 2, 3, 4 ]
```

Создать массив на основе другого объекта, поддерживающего итерации (массивоподобного объекта), можно с помощью статического метода `from()` из класса `Array`:

```
Array.from(<Объект>[, <Функция>[, <this>]])
```

В первом параметре указывается объект, элементы которого будут преобразованы в массив. Во втором параметре можно указать функцию, которая будет вызвана для каждого элемента. Объект, указанный в третьем параметре, будет доступен внутри функции через указатель `this`. Пример:

```
console.log(Array.from( '123' )); // [ '1', '2', '3' ]
console.log(Array.from( [ 1, 2, 3 ],
  function(n) {
    return n * 2;
  }
));                               // [ 2, 4, 6 ]
```

Проверить, является ли объект массивом, позволяет статический метод `isArray()` из класса `Array`. Метод возвращает значение `true`, если объект является массивом, и `false` — в противном случае. Пример:

```
console.log(Array.isArray( [1, 2] )); // true
console.log(Array.isArray( {} ));      // false
```

Отобразить значения массива в табличной форме позволяет метод `table()` объекта `console`:

```
let arr = [1, 2, 3];
console.table(arr);
```

Результат:

(index)	Values
0	1
1	2
2	3

6.2. Получение и изменение элемента массива

Получить значение элемента массива можно, указав его индекс в квадратных скобках. Не забываем, что нумерация элементов массива начинается с 0. Пример:

```
let arr = [1, 2, 3];
console.log( arr[0] );    // 1
console.log( arr[1] );    // 2
console.log( arr[2] );    // 3
```

Если элемент с указанным индексом не существует, будет возвращено значение `undefined`:

```
console.log( arr[4] );    // undefined
```

При желании можно добавить новый элемент массива или изменить значение существующего:

```
let arr = [1, 2];
arr[2] = 3;                // Добавление нового элемента
arr[0] = 55;               // Изменение значения существующего элемента
console.log( arr );        // [ 55, 2, 3 ]
```

Если при добавлении указать индекс больше количества элементов массива, то будет создан элемент с указанным индексом, а также промежуточные элементы, имеющие значение `undefined`:

```
let arr = [1, 2];
arr[5] = 3;                // Добавление нового элемента
console.log( arr );        // [ 1, 2, undefined, undefined, undefined, 3 ]
```

Для сохранения значений элементов массива в переменных можно использовать операцию распаковки массива. В этом случае массив указывается справа от оператора `=`, а переменные внутри квадратных скобок — слева от него. Пример:

```
let arr = [1, 2];
let [a, b] = arr;
console.log( a + ' ' + b );    // 1 2
```

Число элементов массива может не совпадать с количеством переменных внутри квадратных скобок. Кроме того, если элемент не нужен, то достаточно указать переменные только в требуемых позициях. Пример:

```
let a = 0, b = 0, arr = [1, 2, 3, 4, 5];
[a, , b] = arr;
console.log( a + ' ' + b );    // 1 3
```

Если в квадратных скобках указано переменных больше, чем в массиве содержится элементов, "лишние" переменные в качестве значений получат `undefined`:

```
let arr = [1, 2];
[a, b, c] = arr;
console.log(a, b, c);          // 1 2 undefined
```

Однако у "лишних" переменных в квадратных скобках можно указать значения по умолчанию, поставив их после знака `=`. Эти значения переменные получают, если им не "хватит" элементов присваиваемого массива. Пример:

```
let arr = [1, 2];
[a, b, c = 3] = arr;
console.log(a, b, c);          // 1 2 3
```

Значения отдельных элементов массива можно сохранить в переменных, а оставшиеся элементы добавить в массив, указав перед ним оператор `SPREAD`:

```
let arr = [1, 2, 3, 4, 5];
let [a, b, ...other] = arr;
console.log( a + ' ' + b );    // 1 2
console.log( other );           // [ 3, 4, 5 ]
```

С помощью метода `fill()` можно заполнить весь массив или только его часть одинаковым значением. Формат метода:

```
fill(<Значение>[, <Начало>[, <Конец>]])
```

В первом параметре указывается заполняющее значение. В параметре `<Начало>` можно указать начальный индекс диапазона (значение по умолчанию: 0), а в параметре `<Конец>` — конечный индекс (по умолчанию: размер массива). Пример:

```
let arr = new Array(5);
arr.fill(1);
console.log(arr);              // [ 1, 1, 1, 1, 1 ]
arr.fill(2, 1, 4);
console.log(arr);              // [ 1, 2, 2, 2, 1 ]
```

Метод `copyWithin()` копирует заданные элементы ("источник") текущего массива на новое место ("цель") в том же массиве. Формат метода:

```
copyWithin(<Начальный индекс позиции цели>,
           <Начальный индекс позиции источника>[,
           <Конечный индекс позиции источника>])
```

Отметим, что элемент с индексом, указанным третьим параметром, скопирован не будет. Если третий параметр не указан, будут копироваться все элементы источника до конца массива. Метод изменяет текущий массив и не возвращает результата. Пример:

```
let arr = [1, 2, 3, 4, 5, 6, 7];
arr.copyWithin(0, 4);
console.log( arr );           // [ 5, 6, 7, 4, 5, 6, 7 ]
arr = [1, 2, 3, 4, 5, 6, 7];
arr.copyWithin(0, 4, 6);
console.log( arr );           // [ 5, 6, 3, 4, 5, 6, 7 ]
```

Можно указать отрицательные индексы — тогда элементы будут отсчитываться с конца массива:

```
let arr = [1, 2, 3, 4, 5, 6, 7];
arr.copyWithin(0, -3, -1);
console.log( arr );           // [ 5, 6, 3, 4, 5, 6, 7 ]
```

6.3. Определение размера массива

Получить размер массива позволяет свойство `length` массива:

```
let arr = [ 'Один', 'Два', 'Три' ];
console.log(arr.length); // 3
```

Значение свойства `length` можно изменять, тем самым увеличивая или уменьшая размер массива:

```
let arr = [1, 2, 3];
console.log(arr.length); // 3
arr.length = 5;
console.log(arr.length); // 5
console.log(arr);         // [ 1, 2, 3, undefined, undefined ]
arr.length = 2;
console.log(arr.length); // 2
console.log(arr);         // [ 1, 2 ]
```

6.4. Многомерные массивы

Многомерные массивы можно создать перечислением:

```
let arr = new Array(new Array('Один', 'Два', 'Три'),
                    new Array('Четыре', 'Пять', 'Шесть'));
console.log(arr[0][1]); // Два
let arr2 = [
    [ 'Один', 'Два', 'Три' ],
    [ 'Четыре', 'Пять', 'Шесть' ]
];
console.log(arr2[1][1]); // Пять
```

или поэлементно:

```
let arr = new Array();
arr[0] = new Array();
```



```
arr[1] = new Array();
arr[0][0] = 'Один';
arr[0][1] = 'Два';
arr[0][2] = 'Три';
arr[1][0] = 'Четыре';
arr[1][1] = 'Пять';
arr[1][2] = 'Шесть';
console.log(arr[1][2]); // Шесть
let arr2 = [];
arr2[0] = [];
arr2[1] = [];
arr2[0][0] = 'Один';
arr2[0][1] = 'Два';
arr2[0][2] = 'Три';
arr2[1][0] = 'Четыре';
arr2[1][1] = 'Пять';
arr2[1][2] = 'Шесть';
console.log(arr2[0][0]); // Один
```

Обращение к элементу многомерного массива осуществляется с помощью двух индексов:

```
console.log(arr[1][2]); // Шесть
```

Любому элементу массива можно присвоить другой массив, что позволяет создавать так называемые *"зубчатые" многомерные массивы*:

```
let arr = [1, 2, 3];
arr[1] = [2, 3];
arr[2] = [4, 5, 6];
for (let i = 0; i < arr.length; i++) {
  console.log(arr[i]);
}
```

Результат:

```
1
[ 2, 3 ]
[ 4, 5, 6 ]
```

Метод `flat(<Уровень>)` делает текущий массив более "плоским" и возвращает его в качестве результата. В параметре указывается целочисленный уровень, на который следует "поднять" массивы, хранящиеся в текущем массиве (по умолчанию: 1).

Пример:

```
let arr = [1, 2, [3, 4, [5, 6]]];
console.log( arr.flat() );           // [ 1, 2, 3, 4, [ 5, 6 ] ]
console.log( arr.flat(2) );          // [ 1, 2, 3, 4, 5, 6 ]
```

6.5. Создание копии массива

При присваивании значения переменной, хранящей массив, другой переменной в последнюю будет занесена копия ссылки на массив, а не копия самого массива:

```
let arr1, arr2;
arr1 = [1, 2, 3, 4];
arr2 = arr1;           // Присваивается копия ссылки на массив!!!
arr2[0] = 88;
console.log(arr1);     // [ 88, 2, 3, 4 ]
console.log(arr2);     // [ 88, 2, 3, 4 ]
```

Чтобы сделать копию массива, можно, например, воспользоваться методом `slice()` (см. *разд. 6.11*), который возвращает срез массива:

```
let arr1, arr2;
arr1 = [1, 2, 3, 4];
arr2 = arr1.slice(0);
arr2[0] = 88;
console.log(arr1);     // [ 1, 2, 3, 4 ]
console.log(arr2);     // [ 88, 2, 3, 4 ]
```

В случае многомерного массива метод `slice()` создает "поверхностную" копию, а не полную:

```
let arr1, arr2;
arr1 = [ [0, 1], 2, 3, 4 ];
arr2 = arr1.slice(0);
arr2[0][0] = 55;
arr2[1] = 88;
console.log(arr1[0]);  // [ 55, 1 ]
console.log(arr2[0]);  // [ 55, 1 ]
console.log(arr1[1]);  // 2
console.log(arr2[1]);  // 88
```

Как видно из примера, изменение вложенного в `arr2` массива привело к одновременному изменению значения в `arr1`. Иными словами, оба массива содержат ссылки на один и тот же вложенный массив.

Для создания "поверхностной" копии массива можно воспользоваться статическим методом `from()` из класса `Array`:

```
let arr1 = [1, 2, 3, 4], arr2;
arr2 = Array.from(arr1);
arr2[0] = 88;
console.log(arr1);     // [ 1, 2, 3, 4 ]
console.log(arr2);     // [ 88, 2, 3, 4 ]
```

А также оператором `SPREAD`:

```
let arr1 = [1, 2, 3, 4], arr2;
arr2 = [...arr1];
```

```
arr2[0] = 88;
console.log(arr1);    // [ 1, 2, 3, 4 ]
console.log(arr2);    // [ 88, 2, 3, 4 ]
```

6.6. Объединение массивов

Метод `concat(<Список элементов или массивов>)` возвращает массив, полученный в результате объединения текущего массива и списка указанных элементов или других массивов. При этом текущий массив не изменяется. Пример:

```
let arr = [1, 2], arr2 = [3, 4], arr3 = [];
arr3 = arr.concat(arr2);
console.log(arr3);    // [ 1, 2, 3, 4 ]
arr3 = arr.concat(arr2, 5, 6);
console.log(arr3);    // [ 1, 2, 3, 4, 5, 6 ]
```

Можно также воспользоваться оператором `SPREAD`:

```
let arr = [1, 2], arr2 = [3, 4], arr3;
arr3 = [...arr, ...arr2];
console.log(arr3);    // [ 1, 2, 3, 4 ]
```

6.7. Перебор элементов массива

Для перебора элементов массива удобно использовать цикл `for`. Умножим все элементы массива на 2:

```
let arr = [1, 2, 3, 4];
for (let i = 0; i < arr.length; i++) {
    arr[i] *= 2;
}
console.log(arr);    // [ 2, 4, 6, 8 ]
```

Можно также воспользоваться методом `forEach()`. Формат метода:

```
forEach(<Функция>[, <this>])
```

В первом параметре указывается ссылка на функцию, которая будет вызвана для каждого элемента массива, кроме "пустых" элементов. Функция принимает три параметра, через первый будет доступно значение текущего элемента, через второй — его индекс, а через третий — ссылка на массив. Значение, указанное в параметре `<this>`, будет доступно через указатель `this` внутри функции.

Прибавим к значениям всех элементов массива число 2:

```
let arr = [1, 2, 3, 4];
arr.forEach( function(value, index, ar) {
    ar[index] = value + 2;
});
console.log(arr);    // [ 3, 4, 5, 6 ]
```

Пример использования стрелочной функции (указатель `this` изменить нельзя):

```
let arr = [1, 2, 3, 4];
arr.forEach(value => {
  console.log(value);
});
```

Для быстрой вставки метода `forEach()` в редакторе VS Code предназначена следующая комбинация:

```
foreach =>
array.forEach(element => {

});
```

Перебрать элементы массива позволяет также цикл `for...of`. Он имеет следующий формат:

```
for (<Переменная> of <Массив>) {
  <Тело цикла>
}
```

Цикл `for...of` на каждой итерации присваивает `<Переменной>` копию значения элемента массива. Посчитаем сумму всех элементов массива:

```
let arr = [1, 2, 3], n = 0;
for (const value of arr) {
  n += value;
}
console.log(n); // 6
```

6.8. Добавление и удаление элементов массива

Добавить новый элемент массива или изменить значение существующего можно с помощью квадратных скобок:

```
let arr = [1, 2, 3];
arr[3] = 4;
arr[arr.length] = 5; // Добавление в конец массива
console.log(arr);    // [ 1, 2, 3, 4, 5 ]
```

Если при добавлении указать индекс, превышающий число элементов массива, то будет создан элемент с указанным индексом, а также промежуточные элементы, хранящие значения `undefined`:

```
let arr = [1, 2];
arr[5] = 3; // Добавление нового элемента
console.log(arr); // [ 1, 2, undefined, undefined, undefined, 3 ]
```

Можно изменять значение свойства `length`, тем самым увеличивая или уменьшая размер массива:

```
let arr = [1, 2, 3];
console.log(arr.length); // 3
```

```
arr.length = 5;
console.log(arr.length); // 5
console.log(arr);        // [ 1, 2, 3, undefined, undefined ]
arr.length = 2;
console.log(arr.length); // 2
console.log(arr);        // [ 1, 2 ]
```

Удалить значение произвольного элемента массива позволяет оператор `delete`. После удаления значения элемент будет хранить значение `undefined`. Пример:

```
let arr = [1, 2, 3];
delete arr[1];
console.log(arr);    // [ 1, undefined, 3 ]
```

Для добавления и удаления элементов можно пользоваться следующими методами:

- `push(<Элементы через запятую>)` — добавляет указанные элементы в конец текущего массива и возвращает его новый размер:

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

Для добавления элементов из какого-либо массива в конец текущего массива можно воспользоваться следующим кодом:

```
let arr = [1, 2, 3], arr2 = [4, 5];
Array.prototype.push.apply(arr, arr2);
console.log(arr); // [ 1, 2, 3, 4, 5 ]
```

Альтернативный способ с созданием нового массива:

```
let arr = [1, 2, 3], arr2 = [4, 5];
console.log( [...arr, ...arr2] );
// [ 1, 2, 3, 4, 5 ]
```

- `unshift(<Элементы через запятую>)` — добавляет указанные элементы в начало текущего массива и возвращает его новый размер:

```
let arr = [1, 2, 3];
console.log(arr.unshift(4, 5)); // 5
console.log(arr);              // [ 4, 5, 1, 2, 3 ]
```

Альтернативный способ с созданием нового массива:

```
let arr = [1, 2, 3];
console.log( [...[4, 5], ...arr] );
// [ 4, 5, 1, 2, 3 ]
```

- `shift()` — удаляет первый элемент текущего массива и возвращает этот элемент (или значение `undefined`, если массив пуст):

```
let arr = [1, 2, 3];
console.log(arr.shift()); // 1
console.log(arr);        // [ 2, 3 ]
```

- ❑ `pop()` — удаляет последний элемент текущего массива и возвращает этот элемент (или значение `undefined`, если массив пуст):

```
let arr = [1, 2, 3];
console.log(arr.pop());           // 3
console.log(arr);                 // [ 1, 2 ]
```

- ❑ `splice(<Начало>, <Количество>[, <Значения через запятую>])` — удаляет из текущего массива заданное количество элементов, начиная с элемента, имеющего указанный в первом параметре индекс, и вставляет на их место заданные значения. Возвращает массив, состоящий из удаленных элементов. Пример:

```
let arr1 = [ 1, 2, 3, 4, 5 ];
let arr2 = arr1.splice(2, 2);
console.log(arr1);           // [ 1, 2, 5 ]
console.log(arr2);           // [ 3, 4 ]
let arr3 = arr1.splice(1, 1, 7, 8, 9);
console.log(arr1);           // [ 1, 7, 8, 9, 5 ]
console.log(arr3);           // [ 2 ]
let arr4 = arr1.splice(1, 0, 2, 3, 4);
console.log(arr1);           // [ 1, 2, 3, 4, 7, 8, 9, 5 ]
console.log(arr4);           // []
```

6.9. "Переворачивание" массива

Метод `reverse()` меняет порядок следования элементов текущего массива на противоположный (переворачивает массив):

```
let arr = [ 1, 2, 3 ];
arr.reverse();
console.log(arr);           // [ 3, 2, 1 ]
```

6.10. Сортировка массива

Отсортировать массив позволяет метод `sort([<Функция сортировки>])`. Если функция сортировки не указана, будет выполнена обычная сортировка (числа — по возрастанию, а символы — по алфавиту). Пример:

```
let arr = [ 'Один', 'Два', 'Три' ];
arr.sort();
console.log(arr);           // [ 'Два', 'Один', 'Три' ]
```

Стандартный порядок сортировки можно изменить, указав соответствующую функцию сортировки. Она должна принимать два параметра — два сравниваемых элемента массива — и возвращать:

- ❑ положительное число — если первый элемент больше второго;
- ❑ отрицательное число — если второй элемент больше первого;
- ❑ 0 — если элементы равны.

Например, стандартная сортировка зависит от регистра символов:

```
let arr = [ 'единица1', 'Единый', 'Единица2' ];
arr.sort();
console.log(arr); // [ 'Единица2', 'Единый', 'единица1' ]
```

В результате мы получим неправильную сортировку, ведь Единица2 и Единый должны стоять позже единица1. Изменим стандартную сортировку на свою сортировку без учета регистра (листинг 6.1).

Листинг 6.1. Сортировка без учета регистра

```
function cmp(str1, str2) { // Сортировка без учета регистра
    str1 = str1.toLowerCase(); // Преобразуем к нижнему регистру
    str2 = str2.toLowerCase(); // Преобразуем к нижнему регистру
    if (str1 > str2) return 1;
    if (str1 < str2) return -1;
    return 0;
}
let arr = [ 'единица1', 'Единый', 'Единица2' ];
arr.sort(cmp); // Имя функции указывается без скобок
console.log(arr); // [ 'единица1', 'Единица2', 'Единый' ]
```

Приводим оба сравниваемых элемента к нижнему регистру и выполняем стандартное сравнение. Обратите внимание, что мы не изменяем регистр самих элементов массива, т. к. работаем с их копиями.

Порядок сортировки можно поменять на противоположный (листинг 6.2), изменив возвращаемые функцией значения. В качестве примера укажем анонимную функцию вместо обычной.

Листинг 6.2. Сортировка без учета регистра в обратном порядке

```
let arr = [ 'единица1', 'Единый', 'Единица2' ];
arr.sort( function(str1, str2) {
    str1 = str1.toLowerCase();
    str2 = str2.toLowerCase();
    if (str1 > str2) return -1;
    if (str1 < str2) return 1;
    return 0;
});
console.log(arr); // [ 'Единый', 'Единица2', 'единица1' ]
```

6.11. Получение части массива

Метод `slice(<Начало>[, <Конец>])` возвращает часть (срез) массива, начиная от индекса `<Начало>` и заканчивая индексом `<Конец>`, исключая элемент с этим индексом. Если второй параметр не указан, возвращаются все элементы до конца массива.

Пример:

```
let arr1 = [ 1, 2, 3, 4, 5 ];
let arr2 = arr1.slice(1, 4);
console.log(arr2);           // [ 2, 3, 4 ]
let arr3 = arr1.slice(2);
console.log(arr3);           // [ 3, 4, 5 ]
```

В первом параметре можно указать отрицательное значение, в этом случае смещение отсчитывается от конца массива. Получим два последних элемента:

```
let arr1 = [ 1, 2, 3, 4, 5 ];
let arr2 = arr1.slice(-2);
console.log(arr2);           // [ 4, 5 ]
```

6.12. Преобразование массива

Преобразовать массив в строку позволяют следующие методы:

- ❑ `join(<Разделитель>)` — возвращает строку, полученную в результате объединения всех элементов массива через заданный разделитель:

```
let arr = [ 'Один', 'Два', 'Три' ];
let str = arr.join(' - ');
console.log(str); // Один - Два - Три
```

Если параметр не указан, в качестве разделителя будет использоваться запятая:

```
let arr = [ 'Один', 'Два', 'Три' ];
let str = arr.join();
console.log(str); // Один,Два,Три
```

- ❑ `toString()` — возвращает строку, содержащую элементы текущего массива, которые отделяются друг от друга запятыми без пробелов:

```
let arr = [ 'Один', 'Два', 'Три' ];
console.log(arr.toString()); // Один,Два,Три
console.log(arr + '');       // Один,Два,Три
```

- ❑ `toLocaleString()` — аналогичен `toString()`, только если элементами массива являются объекты, для преобразования в строку у этих объектов будет вызван метод `toLocaleString()`:

```
let arr = [ new Date(), 1234.45678 ];
console.log(arr.toString());
// Wed Apr 28 2021 14:07:37 GMT+0300 (Москва, стандартное время),
// 1234.45678
console.log(arr.toLocaleString());
// 28.04.2021, 14:07:37, 1 234,457
```

Преобразовать массив в другой массив можно с помощью метода `map(<Функция>[, <this>])`. Заданная в вызове метода функция будет вызвана для каждого элемента массива, с единственным параметром получит значение текущего элемента и

должна вернуть значение соответствующего элемента нового массива. Метод `map()` возвращает новый массив. Пример создания нового массива, который содержит значения элементов исходного массива, умноженные на 2:

```
let arr = [ 1, 2, 3, 4, 5, 6 ];
let arr2 = arr.map( function(value, index, ar) {
    return value * 2;
});
console.log(arr2);           // [ 2, 4, 6, 8, 10, 12 ]
```

Метод `flatMap()` аналогичен `map()`, только дополнительно делает результирующий массив одномерным (его можно рассматривать как комбинацию методов `map()` и `flat()`):

```
let arr = [ 1, 2, 3, 4, 5, 6 ];
let arr2 = arr.map( function(value, index, ar) {
    return [value, value * 2];
});
console.log(arr2);
// [ [ 1, 2 ], [ 2, 4 ], [ 3, 6 ], [ 4, 8 ], [ 5, 10 ], [ 6, 12 ] ]
let arr3 = arr.flatMap( function(value, index, ar) {
    return [value, value * 2];
});
console.log(arr3);           // [ 1, 2, 2, 4, 3, 6, 4, 8, 5, 10, 6, 12 ]
```

Метод `reduce(<Функция>[, <Начальное значение>])` применяет функцию к парам элементов и накапливает результат. В первом параметре указывается ссылка на функцию, которая будет вызвана для каждого элемента массива. Функция должна принимать четыре параметра, через первый параметр будет доступен результат предыдущих вычислений или `<Начальное значение>` при первом вызове, через второй — значение текущего элемента, через третий — его индекс, а через четвертый — ссылка на массив. Функция должна вернуть результат текущих вычислений. Метод `reduce()` возвращает общий результат вычислений. Пример получения суммы всех элементов массива:

```
let arr = [ 1, 2, 3 ];
let sum = arr.reduce( function(value1, value2, index, ar) {
    return value1 + value2;
}, 0);
console.log(sum);           // 6
```

Метод `reduceRight()` аналогичен методу `reduce()`, но перебирает элементы не слева направо, а справа налево. Пример преобразования многомерного массива в одномерный:

```
let arr = [ [1, 2], [3, 4] ];
let sum = arr.reduceRight( function(arr1, arr2, index, ar) {
    return arr1.concat(arr2);
}, []);
console.log(sum);           // [ 3, 4, 1, 2 ]
```

6.13. Поиск элемента в массиве

Выполнить поиск элемента в массиве позволяют следующие методы:

- ❑ `indexOf(<Элемент>[, <Индекс>])` — ищет первое вхождение заданного элемента в массиве. Сравнение выполняется с помощью оператора `===` (строго равно). Возвращает индекс найденного элемента или значение `-1`, если элемент не найден. Во втором параметре можно указать индекс, с которого начнется поиск. Если второй параметр не указан, поиск начинается с начала массива. Пример:

```
let arr = [ 1, 2, 1, 2, 3 ];
console.log( arr.indexOf(2) );      // 1
console.log( arr.indexOf(2, 2) );   // 3
console.log( arr.indexOf(3) );      // 4
console.log( arr.indexOf(8) );      // -1
```

- ❑ `lastIndexOf(<Элемент>[, <Индекс>])` — аналогичен `index()`, только ищет последнее вхождение элемента в массиве:

```
let arr = [ 1, 2, 1, 2, 3 ];
console.log( arr.lastIndexOf(2) );  // 3
console.log( arr.lastIndexOf(2, 2) ); // 1
console.log( arr.lastIndexOf(3) );  // 4
console.log( arr.lastIndexOf(8) );  // -1
```

- ❑ `includes(<Элемент>[, <Индекс>])` — возвращает значение `true`, если заданный элемент присутствует в массиве, и `false` — в противном случае. Во втором параметре можно указать индекс элемента, с которого начнется поиск. Пример:

```
let arr = [ 1, 2, 3 ], arr2 = [ 1, 2, -3 ];
console.log( arr.includes(1) );      // true
console.log( arr.includes(1, 1) );   // false
console.log( arr2.includes(3) );     // false
```

- ❑ `every(<Функция>[, <this>])` — возвращает `true`, если все элементы массива соответствуют заданному условию, и `false` — в противном случае. В первом параметре указывается ссылка на функцию, которая будет вызвана для каждого элемента массива. Функция принимает три параметра, через первый будет доступно значение текущего элемента, через второй — его индекс, а через третий — ссылка на массив. Функция должна вернуть значение `true`, если элемент соответствует условию, и `false` — в противном случае. Если функция для всех элементов вернула `true`, метод `every()` также возвращает значение `true`. Если функция вернет `false`, проверка прекращается и метод `every()` возвращает `false`. Значение, указанное во втором параметре, будет доступно внутри функции через указатель `this`. Пример:

```
function test(value, index, ar) {
    return value > 0;
}
let arr = [ 1, 2, 3 ], arr2 = [ 1, 2, -3 ];
```

```
console.log( arr.every(test) );    // true
console.log( arr2.every(test) );   // false
```

- ❑ `some(<Функция>[, <this>])` — возвращает `true`, если хотя бы один элемент массива соответствует заданному условию, и `false` — в противном случае. Если заданная функция вернет `true`, проверка прекращается и метод `some()` возвращает `true`. Если функция для всех элементов вернула `false`, метод `some()` также возвращает `false`. Пример:

```
function test(value, index, ar) {
    return value < 0;
}
let arr = [ 1, 2, 3 ], arr2 = [ 1, 2, -3 ];
console.log( arr.some(test) );    // false
console.log( arr2.some(test) );   // true
```

- ❑ `find(<Функция>[, <this>])` — возвращает значение первого элемента, для которого заданная функция вернула `true`, или значение `undefined`, если функция для всех элементов вернула `false`. Пример:

```
function test(value, index, ar) {
    return value < 0;
}
let arr = [ 1, 2, 3 ], arr2 = [ 1, 2, -3 ];
console.log( arr.find(test) );    // undefined
console.log( arr2.find(test) );   // -3
```

- ❑ `findIndex(<Функция>[, <this>])` — возвращает индекс первого элемента, для которого заданная функция вернула `true`, или значение `-1`, если функция для всех элементов вернула `false`:

```
function test(value, index, ar) {
    return value < 0;
}
let arr = [ 1, 2, 3 ], arr2 = [ 1, 2, -3 ];
console.log( arr.findIndex(test) );    // -1
console.log( arr2.findIndex(test) );   // 2
```

6.14. Фильтрация массива

Метод `filter(<Функция>[, <this>])` выполняет фильтрацию массива. В первом параметре указывается ссылка на функцию, которая будет вызвана для каждого элемента массива. Функция принимает три параметра, через первый будет доступно значение текущего элемента, через второй — его индекс, а через третий — ссылка на массив. Функция должна вернуть значение `true`, если элемент соответствует условию, и `false` — в противном случае. Значение, указанное во втором параметре, будет доступно внутри функции через указатель `this`. Метод возвращает массив элементов, соответствующих условию (для которых функция вернула `true`). Исходный массив не изменяется. Пример:

```
function test(value, index, ar) {  
    return value < 4;  
}  
let arr = [ 1, 2, 3, 4, 5, 6 ];  
let arr2 = arr.filter(test);  
console.log(arr2); // [ 1, 2, 3 ]
```

6.15. Методы, возвращающие итераторы

Следующие методы возвращают специальный объект — *итератор*:

❑ `keys()` — возвращает итератор, содержащий индексы элементов массива:

```
let arr = [ 1, 2, 3 ];  
let iterator = arr.keys();  
console.log(iterator); // Object [Array Iterator] {}  
// Перебор всех элементов  
for (const key of iterator) {  
    process.stdout.write(`${key} `);  
} // 0 1 2  
console.log();  
// Преобразование в массив  
let arrKeys = [...arr.keys()];  
console.log(arrKeys); // [ 0, 1, 2 ]
```

❑ `values()` — возвращает итератор, содержащий значения элементов массива:

```
let arr = [ 1, 2, 3 ];  
let iterator = arr.values();  
// Перебор всех элементов  
for (const value of iterator) {  
    process.stdout.write(`${value} `);  
} // 1 2 3  
console.log();  
// Преобразование в массив  
let arrValues = [...arr.values()];  
console.log(arrValues); // [ 1, 2, 3 ]
```

❑ `entries()` — возвращает итератор, содержащий пары индекс/значение всех элементов массива:

```
let arr = [ 1, 2, 3 ];  
let iterator = arr.entries();  
// Перебор всех элементов  
for (const [key, value] of iterator) {  
    process.stdout.write(`${key} => ${value} `);  
} // 0 => 1 1 => 2 2 => 3  
console.log();  
// Преобразование в массив  
let arrEntries = [...arr.entries()];  
console.log(arrEntries); // [ [ 0, 1 ], [ 1, 2 ], [ 2, 3 ] ]
```

Создать итератор, содержащий значения элементов массива, также можно следующим способом:

```
let arr = [ 1, 2, 3 ];
let iterator = arr[Symbol.iterator]();
console.log(iterator); // Object [Array Iterator] {}
// Перебор всех элементов
for (const value of iterator) {
  process.stdout.write(`${value} `);
} // 1 2 3
```

Цикл `for...of` на каждом шаге вызывает метод `next()` объекта итератора. В принципе, мы можем вызвать метод `next()` и сами. Пример:

```
let arr = [ 1, 2, 3 ];
let iterator = arr.values();
// Перебор всех элементов
console.log(iterator.next().value); // 1
console.log(iterator.next().value); // 2
console.log(iterator.next().value); // 3
console.log(iterator.next().value); // undefined
```

6.16. Ассоциативные массивы

К элементам ассоциативного массива выполняется обращение не по числовому индексу, а по строковому ключу. Ключи ассоциативных массивов в JavaScript на самом деле являются свойствами объекта, поэтому для создания ассоциативного массива лучше использовать класс `Object`. Пример:

```
let arr = new Object();
arr['Один'] = 1;
arr['Два'] = 2;
arr['Три'] = 3;
console.log(arr['Один']); // 1
console.log(arr['Два']); // 2
console.log(arr['Три']); // 3
```

Ассоциативные массивы также используются для доступа к свойствам класса вместо классической точки. Для получения числа элементов массива ранее мы обращались к свойству `length` класса `Array` следующим образом:

```
let arr = [ 'Один', 'Два', 'Три' ];
console.log(arr.length); // 3
```

С помощью ассоциативных массивов обращение к свойству `length` будет выглядеть так:

```
let arr = [ 'Один', 'Два', 'Три' ];
console.log(arr['length']); // 3
```

Ни один из методов класса `Array` не позволяет вывести элементы ассоциативного массива. Кстати, свойство `length` также не работает. По этой причине перебрать все элементы массива с помощью стандартного цикла `for` не получится.

Для этой цели существует специальный цикл `for...in`. Он имеет следующий формат:

```
for (<Переменная> in <Экземпляр класса>) {  
  <Тело цикла>  
}
```

Цикл `for...in` на каждой итерации присваивает <Переменной> имя свойства (ключа), с помощью которого можно получить значение соответствующего элемента ассоциативного массива:

```
let arr = new Object();  
arr['Один'] = 1;  
arr['Два'] = 2;  
arr['Три'] = 3;  
for (let key in arr) {  
  // Переменной key на каждой итерации присваивается  
  // ключ текущего элемента ассоциативного массива  
  console.log(`${key} => ${arr[key]}`);  
}
```

В итоге мы получим следующий результат:

```
Один => 1  
Два => 2  
Три => 3
```

6.17. Класс *Map*: словарь

Для создания ассоциативного массива также можно воспользоваться классом `Map`. Ключами такого массива могут быть не только строки, но и объекты.

6.17.1. Создание объекта

Создать объект позволяет следующий конструктор класса `Map`:

```
<Объект> = new Map([<Объект, поддерживающий итерации>])
```

Если параметр не указан, создается пустой объект:

```
let map1 = new Map();  
console.log(map1); // Map {}
```

В качестве параметра можно указать объект, поддерживающий итерации. Например, двумерный массив, содержащий пары ключ/значение:

```
let arr = [ ['a', 1], ['b', 2] ];  
let map2 = new Map(arr);  
console.log(map2); // Map { 'a' => 1, 'b' => 2 }
```

6.17.2. Добавление элементов и изменение значения

Добавить элемент позволяет метод `set(<Ключ>, <Значение>)`:

```
let map = new Map();
map.set('Один', 1);
map.set('Два', 2);
console.log(map); // Map { 'Один' => 1, 'Два' => 2 }
```

Если ключ уже существует, соответствующее ему значение будет изменено:

```
let map = new Map();
map.set('a', 1);
console.log(map); // Map { 'a' => 1 }
map.set('a', 2);
console.log(map); // Map { 'a' => 2 }
```

Метод `set()` возвращает текущий объект, поэтому мы можем создавать цепочки из вызовов метода:

```
let map = new Map();
map.set('Один', 1).set('Два', 2).set('Три', 3);
console.log(map);
// Map { 'Один' => 1, 'Два' => 2, 'Три' => 3 }
```

6.17.3. Получение значения по ключу

Получить значение по ключу позволяет метод `get(<Ключ>)`. Если ключ не существует, то метод вернет значение `undefined`. Пример:

```
let map = new Map();
map.set('a', 1);
console.log( map.get('a') ); // 1
console.log( map.get('b') ); // undefined
```

6.17.4. Проверка наличия ключа

Метод `has(<Ключ>)` возвращает значение `true`, если ключ существует, и `false` — в противном случае:

```
let map = new Map();
map.set('a', 1);
console.log( map.has('a') ); // true
console.log( map.has('b') ); // false
```

6.17.5. Определение размера массива

Для определения размера массива следует воспользоваться свойством `size`:

```
let map = new Map();
map.set('a', 1).set('b', 2).set('c', 3);
console.log( map.size ); // 3
```

6.17.6. Удаление элементов

Удалить элементы позволяют следующие методы:

- ❑ `delete(<Ключ>)` — удаляет элемент, соответствующий указанному ключу. Возвращает значение `true`, если элемент был удален, и `false` — в противном случае (например, если элемент с таким ключом не существует). Пример:

```
let map = new Map();
map.set('a', 1).set('b', 2).set('c', 3);
console.log( map.delete('a') ); // true
console.log( map.delete('k') ); // false
console.log( map );
// Map { 'b' => 2, 'c' => 3 }
```

- ❑ `clear()` — удаляет все элементы. Результата не возвращает. Пример:

```
let map = new Map();
map.set('a', 1).set('b', 2);
console.log( map ); // Map { 'a' => 1, 'b' => 2 }
map.clear();
console.log( map ); // Map {}
```

6.17.7. Методы класса *Map*, возвращающие итераторы

Следующие методы возвращают специальный объект — итератор:

- ❑ `keys()` — возвращает итератор, содержащий ключи в порядке их добавления:

```
let map = new Map();
map.set('a', 1).set('b', 2);
let iterator = map.keys();
console.log(iterator); // [Map Iterator] { 'a', 'b' }
// Перебор всех элементов
for (const key of iterator) {
  process.stdout.write(`${key} `);
} // a b
console.log();
// Преобразование в массив
let arrKeys = [...map.keys()];
console.log(arrKeys); // [ 'a', 'b' ]
```

- ❑ `values()` — возвращает итератор, содержащий значения:

```
let map = new Map();
map.set('a', 1).set('b', 2);
let iterator = map.values();
// Перебор всех элементов
for (const value of iterator) {
  process.stdout.write(`${value} `);
} // 1 2
console.log();
```



```
// Преобразование в массив
let arrValues = [...map.values()];
console.log(arrValues); // [ 1, 2 ]
```

❑ **entries()** — возвращает итератор, содержащий пары ключ/значение:

```
let map = new Map();
map.set('a', 1).set('b', 2);
let iterator = map.entries();
// Перебор всех элементов
for (const [key, value] of iterator) {
  process.stdout.write(`${key} => ${value} `);
} // a => 1 b => 2
console.log();
// Преобразование в массив
let arrEntries = [...map.entries()];
console.log(arrEntries); // [ [ 'a', 1 ], [ 'b', 2 ] ]
```

6.17.8. Перебор элементов словаря

Перебрать все элементы позволяет цикл `for...of`. На каждой итерации доступен массив из ключа и значения. Пример:

```
let map = new Map();
map.set('a', 1).set('b', 2);
for (const [key, value] of map) {
  process.stdout.write(`${key} => ${value} `);
} // a => 1 b => 2
console.log();
for (const arr of map) {
  console.log(arr);
}
// [ 'a', 1 ]
// [ 'b', 2 ]
```

Можно также воспользоваться методом `forEach(<Функция>[, <this>])`. В первом параметре указывается ссылка на функцию, которая будет вызвана для каждого элемента. Функция принимает три параметра, через первый будет доступно значение текущего элемента, через второй — его ключ, а через третий — ссылка на текущий объект. Значение, указанное в параметре `<this>`, будет доступно через указатель `this` внутри функции. Пример:

```
let map = new Map();
map.set('a', 1).set('b', 2);
map.forEach( function(value, key, m) {
  console.log(`${key} => ${value}`);
});
// a => 1
// b => 2
```

6.18. Класс Set: множество

Множество — это набор уникальных элементов, с которым можно сравнивать другие элементы, чтобы определить, принадлежат ли они этому множеству. Для создания множества можно воспользоваться классом Set.

6.18.1. Создание объекта

Создать объект позволяет следующий конструктор класса Set:

```
<Объект> = new Set([<Объект, поддерживающий итерации>])
```

Если параметр не указан, создается пустой объект:

```
let set1 = new Set();  
console.log(set1); // Set {}
```

В качестве параметра можно указать объект, поддерживающий итерации, например массив.

ОБРАТИТЕ ВНИМАНИЕ

Будут добавлены только уникальные элементы.

Пример:

```
let arr = [ 1, 2, 3, 1, 2, 3 ];  
let set2 = new Set(arr);  
console.log(set2); // Set { 1, 2, 3 }
```

6.18.2. Добавление элементов

Добавить элемент в конец объекта позволяет метод add(<Значение>).

ОБРАТИТЕ ВНИМАНИЕ

Будут добавлены только уникальные элементы.

Пример:

```
let mySet = new Set([1, 2, 3]);  
console.log(mySet); // Set { 1, 2, 3 }  
mySet.add(4);  
mySet.add(2); // Не будет добавлено, т. к. значение не уникально  
console.log(mySet); // Set { 1, 2, 3, 4 }
```

Метод add() возвращает текущий объект, поэтому мы можем создавать цепочки из вызовов метода:

```
let mySet = new Set();  
mySet.add(1).add(2).add(3);  
console.log(mySet); // Set { 1, 2, 3 }
```

Также можно добавить во множество элементы любого объекта, поддерживающего итерации:

```
function updateSet(a, b) {
  for (const value of b) {
    a.add(value);
  }
}

let set1 = new Set([1, 2, 3]);
let set2 = new Set([3, 4, 5]);
updateSet(set1, set2);
console.log(set1); // Set { 1, 2, 3, 4, 5 }
updateSet(set1, [1, 4, 5, 7]);
console.log(set1); // Set { 1, 2, 3, 4, 5, 7 }
```

6.18.3. Проверка наличия элемента

Метод `has(<Значение>)` возвращает `true`, если указанное значение существует в множестве, и `false` — в противном случае:

```
let mySet = new Set([1, 2, 3]);
console.log( mySet.has(3) ); // true
console.log( mySet.has(4) ); // false
```

6.18.4. Определение размера множества

Для определения размера множества следует воспользоваться свойством `size`:

```
let mySet = new Set([1, 2, 3]);
console.log( mySet.size ); // 3
```

6.18.5. Удаление элементов

Удалить элементы позволяют следующие методы:

- ❑ `delete(<Значение>)` — удаляет элемент с указанным значением. Возвращает `true`, если элемент был удален, и `false` — в противном случае (например, если элемент с таким значением не существует). Пример:

```
let mySet = new Set([1, 2, 3]);
console.log( mySet.delete(3) ); // true
console.log( mySet.delete(4) ); // false
console.log( mySet );           // Set { 1, 2 }
```

- ❑ `clear()` — удаляет все элементы. Результата не возвращает. Пример:

```
let mySet = new Set([1, 2, 3]);
console.log( mySet ); // Set { 1, 2, 3 }
mySet.clear();
console.log( mySet ); // Set {}
```

6.18.6. Методы класса **Set**, возвращающие итераторы

Следующие методы возвращают специальный объект — итератор:

- ❑ `values()` — возвращает итератор, содержащий значения в порядке их добавления:

```
let mySet = new Set([2, 3, 1]);
let iterator = mySet.values();
// Перебор всех элементов
for (const value of iterator) {
  process.stdout.write(`${value} `);
} // 2 3 1
console.log();
// Преобразование в массив
let arrValues = [...mySet.values()];
console.log(arrValues); // [ 2, 3, 1 ]
```

Можно также воспользоваться методом `keys()`, который является синонимом метода `values()` (поддерживается в целях совместимости с классом `Map`):

```
let mySet = new Set([2, 3, 1]);
console.log([...mySet.keys()]); // [ 2, 3, 1 ]
```

- ❑ `entries()` — возвращает итератор, содержащий пары значение/значение (в целях совместимости с классом `Map`):

```
let mySet = new Set([2, 3, 1]);
console.log([...mySet.entries()]);
// [ [ 2, 2 ], [ 3, 3 ], [ 1, 1 ] ]
```

6.18.7. Перебор элементов множества

Перебрать все элементы позволяет цикл `for...of`. На каждой итерации доступно текущее значение. Пример:

```
let mySet = new Set([1, 2, 3]);
for (const value of mySet) {
  process.stdout.write(`${value} `);
} // 1 2 3
```

Можно также воспользоваться методом `forEach(<Функция>[, <this>])`. В первом параметре указывается ссылка на функцию, которая будет вызвана для каждого элемента. Функция принимает три параметра, через первый и второй будет доступно значение текущего элемента, а через третий — ссылка на текущий объект. Значение, указанное в параметре `<this>`, будет доступно через указатель `this` внутри функции. Пример:

```
let mySet = new Set([1, 2, 3]);
mySet.forEach( function(value, key, s) {
  process.stdout.write(`${value} `);
}); // 1 2 3
```

6.18.8. Преобразование множества в массив

Создать массив на основе значений текущего множества позволяет статический метод `from()` из класса `Array`:

```
let mySet = new Set([1, 2, 3]);
let arr = Array.from(mySet);
console.log(arr); // [ 1, 2, 3 ]
```

Другой способ сделать это:

```
let mySet = new Set([1, 2, 3]);
let arr = [...mySet.values()];
console.log(arr); // [ 1, 2, 3 ]
```

6.18.9. Операции с двумя множествами

Основные операции с двумя множествами:

□ объединение:

```
function unionSet(a, b) {
  let c = new Set(a);
  for (const value of b) {
    c.add(value);
  }
  return c;
}

let set1 = new Set([1, 2, 3]);
let set2 = new Set([3, 4, 5]);
let set3 = unionSet(set1, set2);
console.log(set3); // Set { 1, 2, 3, 4, 5 }
// Альтернативный вариант
let set4 = new Set([...set1, ...set2]);
console.log(set4); // Set { 1, 2, 3, 4, 5 }
```

□ пересечение:

```
function intersectionSet(a, b) {
  let c = new Set();
  for (const value of b) {
    if (a.has(value)) {
      c.add(value);
    }
  }
  return c;
}

let set1 = new Set([1, 2, 3]);
let set2 = new Set([1, 2, 4]);
```

```
let set3 = intersectionSet(set1, set2);
console.log(set3); // Set { 1, 2 }
// Альтернативный вариант
let set4 = new Set([...set1].filter(v => set2.has(v)));
console.log(set4); // Set { 1, 2 }
```

❑ разница:

```
function differenceSet(a, b) {
  let c = new Set(a);
  for (const value of b) {
    c.delete(value);
  }
  return c;
}

let set1 = new Set([1, 2, 3]);
let set2 = new Set([1, 2, 4]);
let set3 = differenceSet(set1, set2);
console.log(set3); // Set { 3 }
// Альтернативный вариант
let set4 = new Set([...set1].filter(v => !set2.has(v)));
console.log(set4); // Set { 3 }
```

❑ симметрическая разность:

```
function symmetricDifferenceSet(a, b) {
  let _union = new Set([...a, ...b]);
  let _intersection = new Set([...a].filter(v => b.has(v)));
  return new Set(
    [..._union].filter(v => !_intersection.has(v))
  );
}

let set1 = new Set([1, 2, 3]);
let set2 = new Set([1, 2, 4]);
let set3 = symmetricDifferenceSet(set1, set2);
console.log(set3); // Set { 3, 4 }
```



ГЛАВА 7

Строки

Строка — это упорядоченная последовательность произвольных символов в кодировке UTF-16. Длина строки ограничена лишь объемом оперативной памяти компьютера. Следует учитывать, что символы внутри строки изменить нельзя, поэтому все строковые методы в качестве значения возвращают новую строку.

7.1. Инициализация строк

В языке JavaScript для хранения строк предназначен тип данных `string`:

```
let str = 'Строка';
console.log(str);           // Строка
console.log(typeof str);    // string
```

Строка заключается в одинарные или двойные кавычки. Одинарную кавычку набрать с клавиатуры проще, поэтому мы будем применять именно этот способ. Хотя можем указать и двойные кавычки, особенно если внутри строки нужно вывести одинарную кавычку. Пример:

```
let str = "Д'Арк";
console.log(str);           // Д'Арк
console.log(typeof str);    // string
```

Разместить строковый литерал на нескольких строках нельзя. Переход на новую строку вызовет синтаксическую ошибку. Пример:

```
let str = 'Строка1
Строка2';                   // Так нельзя!
```

В этом случае следует использовать конкатенацию строк (между литералами указывается символ `+`):

```
let str = 'Строка1' +
          'Строка2';        // Правильно
console.log(str);           // Строка1Строка2
```

Можно также в конце каждой строки указать обратный слеш:

```
let str = 'Строка1\
Строка2';           // Правильно
console.log(str);    // Строка1Строка2
```

Создать строку можно также с помощью класса `String` по следующей схеме:

```
<Экземпляр класса> = new String(<Значение>)
```

Пример:

```
let str = new String('Строка');
console.log(typeof str); // object
```

При создании экземпляра класса `String` тип данных будет `object`, а не `string`. Чтобы получить обычную строку, нужно воспользоваться методом `toString()` или `valueOf()`:

```
let str = new String('Строка');
let str2 = str.toString();
console.log(str2);           // Строка
console.log(typeof str2);    // string
console.log(typeof str.valueOf()); // string
```

Таким способом создания строки пользуются редко. Все методы класса `String` доступны и при использовании элементарного типа. Преобразование элементарного типа в объектный выполняется автоматически. Пример:

```
let str = 'Строка';
let str2 = str.toUpperCase(); // Перевод символов в верхний регистр
console.log(str2);           // СТРОКА
console.log(typeof str2);    // string
```

При использовании метода `toUpperCase()` строка, имеющая тип данных `string`, автоматически преобразуется в экземпляр класса `String`. Затем производится изменение строки (в нашем случае перевод символов в верхний регистр) и возвращается строка, имеющая тип данных `string`. Таким образом, класс `String` является классом-"оберткой" над элементарным типом `string`.

Преобразовать значение в строку можно с помощью следующего формата:

```
<Переменная> = String(<Значение>)
```

Пример:

```
let str = String(10);
console.log(str);           // 10
console.log(typeof str);    // string
```

Статический метод `fromCharCode(<Коды символов через запятую>)` из класса `String` позволяет создать строку из указанных кодов символов в кодировке UTF-16:

```
let str = String.fromCharCode(1055, 1088, 1080, 1074, 1077, 1090);
console.log(str);           // Привет
console.log(typeof str);    // string
```


7.2. Специальные символы в строке

Специальные символы — это комбинации знаков, обозначающих служебные или непечатаемые символы, которые невозможно вставить в строку обычным способом. Перечислим специальные символы, доступные в языке JavaScript:

- `\n` — перевод строки;
- `\r` — возврат каретки;
- `\f` — перевод страницы;
- `\t` — знак табуляции;
- `\v` — знак вертикальной табуляции;
- `\b` — возврат на один символ ("забой");
- `\'` — апостроф;
- `\"` — кавычка;
- `\\` — обратная косая черта.

С помощью специального символа `\n` (перевод строки) мы можем разбить сообщение на несколько строк при выводе в консоли:

```
let str = 'Строка1\nСтрока2\n\nСтрока4';
console.log(str);
// Строка 1
// Строка 2
//
// Строка 4
```

Символ кавычки необходимо обязательно экранировать внутри строки в двойных кавычках с помощью слеша, иначе кавычка станет концом строки. Внутри строки в одинарных кавычках необходимо экранировать апостроф. Пример:

```
console.log("Группа \"Кино\""); // Группа "Кино"
console.log('Д\'Арк');          // Д'Арк
// Так удобнее
console.log('Группа "Кино"');    // Группа "Кино"
console.log("Д'Арк");           // Д'Арк
```

Также следует экранировать сам символ слеша, чтобы вставить его в строку:

```
console.log('строка \'); // Ошибка!
```

Правильно будет так:

```
console.log('строка \\'); // строка \
```

С помощью последовательности `\uNNNN` можно вставить в строку любой символ в кодировке Unicode:

```
console.log('\u005B'); // [
console.log('\u005D'); // ]
```

Кодировка Unicode содержит не только буквы и цифры, но и различные символы, которые можно использовать на веб-страницах вместо изображений. Выведем символ в виде ножниц (в консоли Windows такие символы отображаются в виде прямоугольников, т. к. задана кодировка Windows-866, а не Unicode):

```
console.log('\u2702');      // Символ "ножницы"
```

Для вставки символа Unicode применяется также последовательность `\u{X}`, где `X` — от одной до шести шестнадцатеричных цифр:

```
console.log('\u{5B}');      // [
console.log('\u{5D}');      // ]
```

С помощью последовательности `\xNN` можно вставить в строку символ в кодировке ASCII (указывается шестнадцатеричный код символа):

```
console.log('\x5B');        // [
console.log('\x5D');        // ]
```

7.3. Строки в обратных кавычках (шаблоны строк)

Строка может быть указана внутри обратных апострофов. В этом случае существует возможность выполнить подстановку значения переменной или выражения следующим образом:

```
${<Переменная или выражение>}
```

В начале идет символ `$`, за которым внутри фигурных скобок указывается переменная или выражение:

```
let x = 10;
let str = `x = ${x}`;
console.log(str);           // x = 10
console.log(typeof str);    // string
console.log(`${10 + 2}`);   // 12
```

Кроме того, поддерживается многострочный режим:

```
let str = `Строка 1
Строка 2
Строка 3`;
console.log(str);
// Строка 1
// Строка 2
// Строка 3
```

Обратную кавычку и символ `$`, если не нужно выполнять подстановку, внутри шаблона строки необходимо экранировать с помощью слеша:

```
let str = `str \` str`;
console.log( str );      // str ` str
console.log(`${10 + 2}`); // ${10 + 2}
```

7.4. Неформатированные строки

Статический метод `raw()` из класса `String` в сочетании со строками внутри обратных кавычек позволяет создавать неформатированные строки, внутри которых не нужно экранировать специальные символы. Все специальные символы в таких строках выводятся как есть (например, символ `\n` не будет преобразован в перевод строки, а будет считаться последовательностью символов `\` и `n`). Пример:

```
let str = String.raw`Строка1\nСтрока2`;
console.log( str );    // Строка1\nСтрока2
```

Такие неформатированные строки удобно использовать в шаблонах регулярных выражений, а также при указании пути к файлу или каталогу:

```
let str = String.raw`C:\temp\new\file.txt`;
console.log( str );    // C:\temp\new\file.txt
```

В противном случае все слэши в пути необходимо экранировать:

```
let str = 'C:\\temp\\new\\file.txt';
console.log( str );    // C:\temp\new\file.txt
```

7.5. Конкатенация строк

Для объединения строк (конкатенации) предназначен оператор `+`. Для чисел этот оператор выполняет сложение, но если с одной из сторон от оператора `+` находится строка, то второй операнд преобразуется в строку, а не в число. Пример:

```
let str = 'Строка1' + 'Строка2';
console.log( str ); // Строка1Строка2
```

Конкатенацию иногда приходится использовать, чтобы разбить длинную строку на несколько строк:

```
let str = 'Строка 1 ' +
'Строка 2 ' +
'Строка 3';
console.log(str); // Строка 1 Строка 2 Строка 3
```

Часто необходимо сформировать строку, состоящую из имени переменной и ее значения. Для этого подойдет конкатенация. Пример:

```
let x = 10;
let str = 'Значение равно ' + x;
console.log( str ); // Значение равно 10
```

Еще один способ заключается в использовании строк внутри обратных кавычек:

```
let x = 10;
let str = `Значение равно ${x}`;
console.log( str ); // Значение равно 10
```

Для объединения строк также можно воспользоваться оператором `+=`:

```
let str = 'Строка1';
str += 'Строка2';
console.log( str ); // Строка1Строка2
```

Наконец, объединить указанные строки можно посредством метода `concat` (<Строки через запятую>):

```
let str1 = 'Строка 1';
let str2 = str1.concat('Строка 2', 'Строка 3');
console.log(str2); // Строка 1Строка 2Строка 3
```

7.6. Определение длины строки

Получить длину строки позволяет свойство `length`:

```
let str = 'Строка';
console.log( str.length ); // 6
```

7.7. Обращение к отдельному символу в строке

К символу строки можно обратиться как к элементу массива — указав его индекс в квадратных скобках. Нумерация символов начинается с 0. Можно только получить символ по индексу, изменить символ таким образом нельзя. Пример:

```
let str = 'Строка';
console.log( str[1] ); // т
```

Также можно использовать метод `charAt` (<Индекс символа>):

```
let str = 'Строка';
console.log( str.charAt(0) ); // С
console.log( str.charAt(1) ); // т
```

Если указанный индекс отсутствует в строке, то при использовании квадратных скобок получим значение `undefined`, а метод `charAt()` вернет пустую строку:

```
let str = 'Строка';
console.log( str[6] ); // undefined
console.log( str.charAt(6) ); // <Пустая строка>
```

Метод `charCodeAt` (<Индекс символа>) позволяет получить Unicode-код символа с указанным индексом:

```
let str = 'Строка';
console.log( str.charCodeAt(0) ); // 1057
console.log( str.charCodeAt(1) ); // 1090
console.log( str.charCodeAt(6) ); // NaN
```

Строки поддерживают итерации, поэтому мы можем перебрать все символы в цикле:

```
for (const char of 'Строка') {  
  console.log(char);  
}
```

Получить итератор из строки позволяет следующая инструкция:

```
let iterator = str[Symbol.iterator]();
```

Пример:

```
let str = 'str';  
let iterator = str[Symbol.iterator]();  
console.log( iterator.next().value ); // s  
console.log( iterator.next().value ); // t  
console.log( iterator.next().value ); // r  
console.log( iterator.next().value ); // undefined
```

Можно также преобразовать строку в массив, в котором каждый элемент будет содержать только одну букву:

```
let arr = Array.from('Строка');  
console.log(arr); // [ 'С', 'т', 'р', 'о', 'к', 'а' ]
```

Как говорилось ранее, строки в языке JavaScript являются последовательностями символов в кодировке UTF-16. На сегодняшний день 16 битов недостаточно для кодирования символов всех языков мира. Поэтому некоторые символы могут кодироваться 32 битами. Буквы кириллицы не попадают в диапазон таких кодов, поэтому мы не станем рассматривать методы, предназначенные для работы с *кодowymi точками*. Однако знать об этом нужно. В случае необходимости вы всегда сможете найти описание этих методов в документации.

7.8. Изменение регистра символов

Для изменения регистра символов предназначены следующие методы:

- ❑ `toLowerCase()` — преобразует символы к нижнему регистру:

```
let str = 'строка СТРОКА СтРоКа';  
console.log( str.toLowerCase() ); // строка строка строка
```

- ❑ `toUpperCase()` — преобразует символы к верхнему регистру:

```
let str = 'строка СТРОКА СтРоКа';  
console.log( str.toUpperCase() ); // СТРОКА СТРОКА СТРОКА
```

- ❑ `toLocaleLowerCase()` — преобразует символы к нижнему регистру, учитывая настройки локали:

```
let str = 'строка СТРОКА СтРоКа';  
console.log( str.toLocaleLowerCase() ); // строка строка строка
```

- ❑ `toLocaleUpperCase()` — преобразует символы к верхнему регистру, учитывая настройки локали:

```
let str = 'строка СТРОКА СтРоКа';  
console.log( str.toLocaleUpperCase() ); // СТРОКА СТРОКА СТРОКА
```

7.9. Получение фрагмента строки

Получить фрагмент строки позволяют следующие методы:

- ❑ `substr(<Начало фрагмента>[, <Длина фрагмента>])` — возвращает фрагмент строки заданной длины. Если второй параметр пропущен, возвращаются все символы до конца строки. Пример:

```
let str = 'строка';  
console.log( str.substr(0, 1) ); // с  
console.log( str.substr(1) ); // трока
```

- ❑ `substring(<Начало фрагмента>[, <Конец фрагмента>])` — возвращает фрагмент строки, заданный индексами начального и конечного символов. Последний символ во фрагмент не включается. Если второй параметр пропущен, возвращаются все символы до конца строки. Пример:

```
let str = 'строка';  
console.log( str.substring(0, 1) ); // с  
console.log( str.substring(1, 4) ); // тро  
console.log( str.substring(1) ); // трока
```

- ❑ `slice(<Начало фрагмента>[, <Конец фрагмента>])` — возвращает фрагмент строки, заданный индексами начального и конечного символов. Последний символ во фрагмент не включается. Пример:

```
let str = 'строка';  
console.log( str.slice(0, 1) ); // с  
console.log( str.slice(1, 4) ); // тро  
console.log( str.slice(1) ); // трока
```

Метод `slice()`, в отличие от `substring()`, позволяет указывать отрицательные индексы. Значение такого индекса вычитается из длины строки. Пример:

```
let str = 'строка';  
console.log( str.substring(-2) ); // строка  
console.log( str.slice(-2) ); // ка  
console.log( str.substring(1, -2) ); // с  
console.log( str.slice(1, -2) ); // тро
```

7.10. Сравнение строк

Для сравнения строк можно использовать операторы сравнения:

```
console.log('строка' == 'строка'); // true  
console.log('строка1' > 'строка2'); // false  
console.log('строка1' < 'строка2'); // true
```

Сравнение зависит от регистра символов. Кроме того, следует учитывать, что буква "ё" не входит в диапазон от "а" до "я":

```
console.log( 'я' > 'ё' );           // false
console.log( 'я'.charCodeAt(0) );   // 1103
console.log( 'ё'.charCodeAt(0) );   // 1105
console.log( 'е'.charCodeAt(0) );   // 1077
console.log( 'ж'.charCodeAt(0) );   // 1078
```

Для сравнения строк с учетом настроек локали следует воспользоваться методом `str1.localeCompare(str2)`. Метод возвращает:

- положительное число — если `str1` больше `str2`;
- отрицательное число — если `str1` меньше `str2`;
- 0 — если строки равны.

Пример:

```
console.log( 'я'.localeCompare('ё') ); // 1
console.log( 'ё'.localeCompare('я') ); // -1
console.log( 'я'.localeCompare('я') ); // 0
```

7.11. Поиск и замена в строке

Для поиска и замены в строке предусмотрены следующие методы:

- `includes(<Подстрока>[, <Начальная позиция поиска>])` — возвращает значение `true`, если указанная подстрока входит в текущую строку, и `false` — в противном случае. Если второй параметр не задан, поиск начинается с начала строки. Пример:

```
let str = 'строка строка';
console.log( str.includes('стр') ); // true
console.log( str.includes('стр', 3) ); // true
console.log( str.includes('Стр') ); // false
```

- `indexOf(<Подстрока>[, <Начальная позиция поиска>])` — возвращает индекс первого вхождения заданной подстроки в текущей строке. Если второй параметр не задан, поиск начинается с начала строки. Если подстрока не найдена, возвращается значение `-1`. Пример:

```
let str = 'строка строка';
console.log( str.indexOf('стр') ); // 0
console.log( str.indexOf('стр', 3) ); // 7
console.log( str.indexOf('Стр') ); // -1
if (str.indexOf('стр') !== -1) {
  console.log('Найдено');
}
```

- `lastIndexOf(<Подстрока>[, <Начальная позиция поиска>])` — возвращает индекс последнего вхождения заданной подстроки в текущей строке. Поиск ведется от конца к началу. Если подстрока не найдена, возвращается значение `-1`. Пример:

```
let str = 'строка строка';
console.log( str.lastIndexOf('стр') );    // 7
console.log( str.lastIndexOf('стр', 6) );  // 0
console.log( str.lastIndexOf('Стр') );    // -1
```

- ❑ `startsWith(<Подстрока>[, <Начальная позиция поиска>])` — возвращает `true`, если строка начинается с заданной подстроки, и `false` — в противном случае:

```
let str = 'строка';
console.log( str.startsWith('стр') );    // true
console.log( str.startsWith('ока', 3) ); // true
console.log( str.startsWith('Стр') );    // false
```

- ❑ `endsWith(<Подстрока>[, <Длина>])` — возвращает `true`, если строка заканчивается заданной подстрокой, и `false` — в противном случае:

```
let str = 'строка';
console.log( str.endsWith('ока') );      // true
console.log( str.endsWith('стр', 3) );   // true
console.log( str.endsWith('окА') );      // false
```

- ❑ `padStart(<Длина>[, <Символы>])` — добавляет пробелы или символы из второго параметра в начало строки таким образом, чтобы длина новой строки равнялась значению из первого параметра, и возвращает результат:

```
let str = 'str';
console.log('' + str.padStart(10) + ''); // '      str'
console.log('' + str.padStart(10, '+') + ''); // '++++++str'
console.log('' + str.padStart(10, 'abc') + ''); // 'abcbacstr'
```

- ❑ `padEnd(<Длина>[, <Символы>])` — добавляет пробелы или символы из второго параметра в конец строки таким образом, чтобы длина новой строки равнялась значению из первого параметра, и возвращает результат:

```
let str = 'str';
console.log('' + str.padEnd(10) + '');    // 'str      '
console.log('' + str.padEnd(10, '+') + ''); // 'str++++++'
console.log('' + str.padEnd(10, 'abc') + ''); // 'strabcbca'
```

- ❑ `trim()` — удаляет пробельные символы в начале и конце строки и возвращает результат:

```
let str = ' строка \n\t\v\r';
console.log('' + str.trim() + '');        // 'строка'
```

- ❑ `trimLeft()` — удаляет пробельные символы в начале строки и возвращает результат:

```
let str = '  строка  ';
console.log('' + str.trimLeft() + '');    // 'строка  '
```

- ❑ `trimRight()` — удаляет пробельные символы в конце строки и возвращает результат:


```
let str = '   строка   ';  
console.log("'" + str.trimRight() + "'"); // '   строка'
```

- ❑ `search(<Регулярное выражение>)` — возвращает индекс первого вхождения подстроки, совпадающей с указанным регулярным выражением;
- ❑ `match(<Регулярное выражение>)` — возвращает массив с результатами поиска первой подстроки, совпадающей с заданным регулярным выражением;
- ❑ `matchAll(<Регулярное выражение>)` — возвращает итератор по результатам поиска всех подстрок, совпадающих с заданным регулярным выражением;
- ❑ `replace(<Регулярное выражение>, <Текст для замены>)` — возвращает строку, которая является результатом замены в исходной строке первой найденной подстроки, совпадающей с заданным регулярным выражением;
- ❑ `replaceAll(<Регулярное выражение>, <Текст для замены>)` — возвращает строку, которая является результатом замены в исходной строке всех подстрок, совпадающих с заданным регулярным выражением.

Примеры использования последних пяти методов мы рассмотрим при изучении регулярных выражений и встроенного класса `RegExp` (см. *разд. 8.2*).

7.12. Преобразование строки в массив

Метод `split(<Разделитель>[, <Лимит>])` возвращает массив, полученный в результате деления строки на подстроки по заданному разделителю. Вторым параметром можно указать максимальное число элементов в результирующем массиве.

Пример:

```
let str = 'A\\tB\\tC\\tD';  
let arr = str.split('\\t');  
console.log(arr); // [ 'A', 'B', 'C', 'D' ]  
arr = str.split('\\t', 3);  
console.log(arr); // [ 'A', 'B', 'C' ]
```

Если в первом параметре указать пустую строку, каждый элемент массива будет содержать по одной букве:

```
let str = 'строка';  
let arr = str.split('');  
console.log(arr); // [ 'с', 'т', 'р', 'о', 'к', 'а' ]
```

Для получения массива символов можно также воспользоваться статическим методом `from()` из класса `Array`:

```
let arr = Array.from('строка');  
console.log(arr); // [ 'с', 'т', 'р', 'о', 'к', 'а' ]
```

7.13. URL-кодирование строк

Выполнить URL-кодирование строки позволяют следующие глобальные функции:

- ❑ `encodeURIComponent(<URL-адрес>)` — кодирует URL-адрес целиком и возвращает результат;
- ❑ `decodeURIComponent(<Строка>)` — декодирует строку, закодированную функцией `encodeURIComponent()`, и возвращает результат:

```
let url = 'test.php?id=5&n=тест';
let str = encodeURIComponent(url);
console.log(str); // test.php?id=5&n=%D1%82%D0%B5%D1%81%D1%82
console.log( decodeURIComponent(str) ); // test.php?id=5&n=тест
```

- ❑ `encodeURIComponent(<Строка>)` — выполняет URL-кодирование строки и возвращает результат:

```
let str = encodeURIComponent('Строка');
console.log(str); // %D0%A1%D1%82%D1%80%D0%BE%D0%BA%D0%B0
console.log( decodeURIComponent(str) ); // Строка
```

В отличие от функции `encodeURIComponent()`, заменяет все спецсимволы шестнадцатеричными кодами:

```
let url = 'test.php?n=тест';
let str = encodeURIComponent(url);
console.log(str); // test.php%3Fn%3D%D1%82%D0%B5%D1%81%D1%82
console.log( decodeURIComponent(str) ); // test.php?n=тест
```

- ❑ `decodeURIComponent(<Строка>)` — декодирует строку, закодированную функцией `encodeURIComponent()`, и возвращает результат.

ПРИМЕЧАНИЕ

Перечисленные ранее функции в случае ошибки могут сгенерировать исключение, поэтому кодирование и особенно декодирование лучше выполнять внутри инструкции `try...catch`.

7.14. Повтор строки

Метод `repeat(<Количество повторов>)` позволяет создать строку, состоящую из указанного количества повторов исходной строки:

```
let str = 'abc';
console.log( str.repeat(1) ); // abc
console.log( str.repeat(2) ); // abcabc
console.log( str.repeat(3) ); // abcabcabc
```

7.15. Выполнение команд, содержащихся в строке

Глобальная функция `eval(<Строка>)` выполняет выражение JavaScript, хранящееся в строке, и возвращает результат его выполнения:

```
let str = '3 + 5';
let x = eval(str);
console.log(x);           // 8
console.log(typeof x);    // number
```

Переменные, объявленные внутри выражения, в обычном режиме импортируются в область видимости указателя `this` внутри функции `eval()`. Если функция `eval()` вызывается непосредственно внутри пользовательской функции, то будут перезаписаны значения локальных переменных. А если функция вызывается через ссылку (указатель `this` при этом ссылается на глобальную область видимости), то могут быть затронуты значения глобальных переменных. Пример:

```
var x = 10;
let str = 'var x = 3;';
eval(str);
console.log(x);           // 3
```

Если используется ключевое слово `let`, то значение не перезаписывается:

```
let x = 10;
let str = 'let x = 3;';
eval(str);
console.log(x);           // 10
```

В строгом режиме переменные не импортируются вообще:

```
'use strict';
var x = 10;
let str = 'var x = 3; x';
let y = eval(str);
console.log(x);           // 10
console.log(y);           // 3
```

ВНИМАНИЕ!

Никогда не передавайте функции `eval()` строку из неизвестного источника. В противном случае злоумышленник сможет выполнить любой код. Старайтесь вообще избегать использования этой функции в Node.js.



Регулярные выражения

Регулярное выражение — это своего рода шаблон для поиска в строке совпадающих с ним подстрок. Поддержку регулярных выражений в JavaScript реализует класс `RegExp`.

8.1. Создание регулярного выражения

Создать регулярное выражение на основе заданного шаблона можно двумя способами:

```
<Регулярное выражение> = new RegExp(<Шаблон>[, <Модификатор>])
<Регулярное выражение> = /<Шаблон>/[<Модификатор>]
```

Необязательный параметр `<Модификатор>` задает дополнительные параметры поиска. Он может содержать следующие символы (или их комбинацию):

- ❑ `i` — поиск без учета регистра;
- ❑ `g` — глобальный поиск (поиск всех совпадающих подстрок);
- ❑ `m` — многострочный режим. Символ `^` соответствует началу каждой подстроки, а `$` — концу каждой подстроки. Пример:

```
let p = new RegExp('^ [0-9] $', 'mg');
let str = '1\n2\n3\nстрока\n4';
let arr = str.match(p);
console.log(arr); // [ '1', '2', '3', '4' ]
// Преобразование шаблона в строку
console.log( p.toString() ); // /^[0-9]$/gm
```

- ❑ `s` — если флаг указан, метасимвол `.` будет соответствовать любому символу, включая символ перевода строки (`\n`):

```
let p = /. /;
console.log( p.dotAll ); // false
console.log( p.test('\n') ); // false
p = /. /s;
```

```
console.log( p.dotAll );      // true
console.log( p.test('\n') ); // true
```

- ❑ **d** — если флаг указан, массив с результатами поиска будет содержать свойство `indices` с массивом, каждый элемент которого будет представлять собой массив с индексами первого и последнего символов каждой найденной подстроки;
- ❑ **y** — если флаг указан, на совпадение с регулярным выражением будет проверяться лишь подстрока, начальный символ которой имеет индекс, заданный значением свойства `lastIndex` объекта регулярного выражения;
- ❑ **u** — позволяет использовать внутри строки символы Unicode, состоящие из четырех байтов. Внутри шаблона регулярного выражения возможна дополнительная конструкция `\p{...}`. Пример:

```
let p = /\p{Letter}+/gu;
let str = 'abc 123 def';
let arr = str.match(p);
console.log(arr);           // [ 'abc', 'def' ]
console.log( p.unicode );   // true
```

8.2. Методы класса *String*

При изучении класса `String` нами были оставлены без внимания некоторые методы. Рассмотрим их подробно:

- ❑ `search(<Регулярное выражение>)` — возвращает индекс первого вхождения подстроки, совпадающей с заданным регулярным выражением, или значение `-1`, если совпадений нет:

```
let p = /20[14]/;
let str = '200, 201, 202, 203, 204';
console.log( str.search(p) ); // 5
console.log( '200'.search(p) ); // -1
```

Шаблону `20[14]` соответствуют только два числа: 201 и 204.

Можно также воспользоваться следующим синтаксисом:

```
let p = /20[14]/;
let str = '200, 201, 202, 203, 204';
console.log( p[Symbol.search](str) ); // 5
```

- ❑ `match(<Регулярное выражение>)` — возвращает массив с результатами поиска подстрок, совпадающих с заданным регулярным выражением, или значение `null`, если совпадений нет:

```
let p = /20[14]/;
let str = '200, 201, 202, 203, 204';
let arr = str.match(p);
console.log( arr );
```

```
// [  
//   '201',  
//   index: 5,  
//   input: '200, 201, 202, 203, 204',  
//   groups: undefined  
// ]  
console.log( '200'.match(p) ); // null
```

ОБРАТИТЕ ВНИМАНИЕ

При отсутствии флага `g` результат дополнительно содержит свойства `index` (индекс найденного фрагмента) и `input` (оригинальная строка).

Если в регулярном выражении указан модификатор `d`, то возвращенный методом массив будет дополнительно включать свойство `indices`. Оно будет хранить массив, единственный элемент которого представляет собой массив из двух элементов — индексов начального и конечного символов найденной подстроки в текущей строке. Пример:

```
let str = '200, 201, 202, 203, 204';  
let p = /20[14]/d;  
let arr = str.match(p);  
console.log( arr );  
// [  
//   '201',  
//   index: 5,  
//   input: '200, 201, 202, 203, 204',  
//   groups: undefined,  
//   indices: [ [ 5, 8 ], groups: undefined ]  
// ]
```

Предыдущие примеры выведут только 201, т. к. не указан модификатор глобального поиска `g`. Модифицируем шаблон, чтобы получить все вхождения:

```
let p = /20[14]/g;  
let str = '200, 201, 202, 203, 204';  
let arr = str.match(p);  
console.log(arr); // [ '201', '204' ]
```

Теперь будут выведены все подстроки, совпадающие с регулярным выражением.

Можно также воспользоваться следующим синтаксисом:

```
let p = /20[14]/g;  
let str = '200, 201, 202, 203, 204';  
let arr = p[Symbol.match](str);  
console.log(arr); // [ '201', '204' ]
```

- ❑ `matchAll(<Регулярное выражение>)` — возвращает итератор с результатами поиска всех подстрок, совпадающих с заданным регулярным выражением. В отличие от метода `match()`, дополнительно позволяет получить фрагменты (группы), заключенные в шаблоне в круглые скобки. Пример использования этого метода приведен в листинге 8.1.

Листинг 8.1. Использование метода `matchAll()`

```

let p = /(20)([14])/g;
let str = '200, 201, 202, 203, 204';
let iterator = str.matchAll(p);
for (const match of iterator) {
    console.log(match);
}
/*
[
  '201',           // Полное совпадение
  '20',           // Фрагмент внутри первых скобок
  '1',            // Фрагмент внутри вторых скобок
  index: 5,
  input: '200, 201, 202, 203, 204',
  groups: undefined
]
[
  '204',
  '20',
  '4',
  index: 20,
  input: '200, 201, 202, 203, 204',
  groups: undefined
]
*/
// Преобразование в массив
let arr = [...str.matchAll(/20[14]/g)];
console.log(arr);
/*
[
  [
    '201',
    index: 5,
    input: '200, 201, 202, 203, 204',
    groups: undefined
  ],
  [
    '204',
    index: 20,
    input: '200, 201, 202, 203, 204',
    groups: undefined
  ]
]
*/
// Преобразование в массив
let arr2 = Array.from(str.matchAll(/20[14]/g), a => a[0]);
console.log(arr2); // [ '201', '204' ]

```

ОБРАТИТЕ ВНИМАНИЕ

Итератор можно использовать только один раз. Чтобы создать новый итератор, следует вызвать метод `matchAll()` повторно.

Если в регулярном выражении указан модификатор `d`, возвращенный методом массив будет дополнительно включать свойство `indices`. Оно будет хранить массив, каждый элемент которого представляет собой массив из двух элементов — индексов начального и конечного символов очередной найденной подстроки или ее фрагмента в текущей строке. Пример использования модификатора `d` приведен в листинге 8.2.

Листинг 8.2. Пример использования модификатора `d` в методе `matchAll()`

```
let p = /(20)([14])/gd;
let str = '200, 201, 202, 203, 204';
let iterator = str.matchAll(p);
for (const match of iterator) {
  console.log(match);
}
// [
//   '201',
//   '20',
//   '1',
//   index: 5,
//   input: '200, 201, 202, 203, 204',
//   groups: undefined,
//   indices: [ [ 5, 8 ], [ 5, 7 ], [ 7, 8 ], groups: undefined ]
// ]
// [
//   '204',
//   '20',
//   '4',
//   index: 20,
//   input: '200, 201, 202, 203, 204',
//   groups: undefined,
//   indices: [ [ 20, 23 ], [ 20, 22 ], [ 22, 23 ],
//   groups: undefined ]
// ]
```

Можно также воспользоваться следующим синтаксисом:

```
let p = /(20)([14])/g;
let str = '200, 201, 202, 203, 204';
let iterator = p[Symbol.matchAll](str);
let arr = Array.from(iterator, a => a[0]);
console.log(arr); // [ '201', '204' ]
```


- ❑ `replace(<Регулярное выражение>, <Текст для замены>)` — возвращает строку, которая является результатом замены в исходной строке подстроки, совпадающих с заданным регулярным выражением:

```
let p = /20[14]/g;
let str = '200, 201, 202, 203, 204';
let str2 = str.replace(p, '207');
console.log(str2);           // 200, 207, 202, 203, 207
```

В качестве первого параметра можно указать строку, но в этом случае будет проведена замена только первого вхождения подстроки в исходную строку:

```
let str = '200, 201, 200, 200';
let str2 = str.replace('200', '207');
console.log(str2);           // 207, 201, 200, 200
```

В качестве второго параметра можно указать ссылку на функцию. Через первый параметр в функции будет доступна подстрока, полностью соответствующая шаблону, через следующие параметры — подвыражения, которые соответствуют фрагментам, заключенным в шаблоне в круглые скобки, через предпоследний параметр — смещение подстроки внутри строки, а через последний — исходная строка. Функция должна вернуть подстроку для замены.

В качестве примера найдем все числа в строке и прибавим к ним число 10:

```
let p = /([0-9]+)/g;
let str = '200, 201, 202, 203, 204';
let str2 = str.replace(p, function(s, x, offset, sourceStr) {
  console.log(x + ' ' + offset + ' ' + sourceStr);
  let n = parseInt(s, 10);
  n += 10;
  return n + ' ';
});
console.log(str2); // 210, 211, 212, 213, 214
```

В строке для замены можно использовать специальные переменные `$1`, ..., `$N`, через которые доступны фрагменты, заключенные в шаблоне в круглые скобки. Через переменную `&` доступна вся подстрока, через переменную ``` — фрагмент до подстроки, а через переменную `'` — фрагмент после подстроки.

В качестве примера поменяем два тега местами:

```
let p = /<([a-z]+)><([a-z]+)>/g;
let str = '<br><hr>';
let str2 = str.replace(p, '<$2><$1>');
console.log(str2); // <hr><br>
```

Можно также воспользоваться следующим синтаксисом:

```
let p = /<([a-z]+)><([a-z]+)>/g;
let str = '<br><hr>';
let str2 = p[Symbol.replace](str, '<$2><$1>');
console.log(str2); // <hr><br>
```

- ❑ `replaceAll(<Регулярное выражение>, <Текст для замены>)` — возвращает строку, которая является результатом замены в исходной строке всех подстрок, совпадающих с указанным регулярным выражением:

```
let p = /20[14]/g;
let str = '200, 201, 202, 203, 204';
let str2 = str.replaceAll(p, '207');
console.log(str2);           // 200, 207, 202, 203, 207
```

Если в регулярном выражении отсутствует флаг `g`, будет сгенерировано исключение:

```
TypeError: replaceAll must be called with a global RegExp
```

В первом параметре можно указать обычную строку:

```
let str = '200, 201, 200, 200';
let str2 = str.replaceAll('200', '207');
console.log(str2);           // 207, 201, 207, 207
```

В качестве второго параметра можно указать ссылку на функцию. Через первый параметр в функции будет доступна подстрока, полностью соответствующая шаблону, через следующие параметры — подвыражения, которые соответствуют фрагментам, заключенным в шаблоне в круглые скобки, через предпоследний параметр — смещение подстроки внутри строки, а через последний — исходная строка. Функция должна вернуть подстроку для замены.

В качестве примера найдем все числа в строке и прибавим к ним 10:

```
let p = /[0-9]/g;
let str = '200, 201, 202, 203, 204';
let str2 = str.replaceAll(p,
  function(s, x, offset, sourceStr) {
    console.log(x + ' ' + offset + ' ' + sourceStr);
    let n = parseInt(s, 10);
    n += 10;
    return n + ' ';
  });
console.log(str2); // 210, 211, 212, 213, 214
```

В строке для замены можно использовать специальные переменные `$1`, ..., `$N`, через которые доступны фрагменты, заключенные в шаблоне в круглые скобки. Через переменную `&` доступна вся подстрока, через переменную ``` — фрагмент до подстроки, а через переменную `'` — фрагмент после подстроки.

В качестве примера поменяем два тега местами:

```
let p = /<([a-z]+)><([a-z]+)>/g;
let str = '<br><hr>';
let str2 = str.replaceAll(p, '<$2><$1>');
console.log(str2); // <hr><br>
```

Метод `split(<Регулярное выражение>[, <Лимит>])` также поддерживает регулярные выражения. Он возвращает массив, полученный в результате деления строки на

подстроки по фрагменту, совпадающему с заданным регулярным выражением. Вторым параметром можно указать максимальное число элементов в результирующем массиве. Пример:

```
let p = /\s/g;
let str = '1 2 3\n4\t5\r6';
let arr = str.split(p);
console.log(arr);           // [ '1', '2', '3', '4', '5', '6' ]
arr = str.split(p, 3);
console.log(arr);           // [ '1', '2', '3' ]
```

Также можно воспользоваться следующим синтаксисом:

```
let p = /\s/g;
let str = '1 2 3\n4\t5\r6';
let arr = p[Symbol.split](str);
console.log(arr);           // [ '1', '2', '3', '4', '5', '6' ]
arr = p[Symbol.split](str, 3);
console.log(arr);           // [ '1', '2', '3' ]
```

8.3. Методы класса *RegExp*

Вместо методов класса *String* можно воспользоваться методами класса *RegExp*:

- ❑ `test(<Строка>)` — возвращает `true`, если в заданной строке присутствует подстрока, совпадающая с текущим регулярным выражением, и `false` — в противном случае. Если используется глобальный поиск, то при каждом вызове метода указатель текущей позиции будет перемещаться в конец очередной найденной подстроки. Пример:

```
let p = /20[14]/g;
let str = '200, 201, 202, 203, 204';
console.log( p.test(str) ); // true
console.log( p.test(str) ); // true
console.log( p.test(str) ); // false
console.log( p.test('200') ); // false
```

Если в регулярном выражении указан модификатор `y`, на совпадение будет проверяться лишь подстрока, начальный символ которой имеет индекс, заданный значением свойства `lastIndex` объекта регулярного выражения:

```
let p = /20[14]/y;
let str = '220, 201, 202, 203, 204';
console.log( p.test(str) ); // false
p.lastIndex = 5;
console.log( p.test(str) ); // true
```

- ❑ `exec(<Строка>)` — возвращает массив с результатами поиска в заданной строке подстрок, совпадающих с текущим регулярным выражением. Если используется глобальный поиск, то при каждом вызове метода указатель текущей позиции будет перемещаться в конец очередной найденной подстроки. Если совпадений

нет, метод вернет значение `null`. Пример использования этого метода приведен в листинге 8.3.

Листинг 8.3. Использование метода `exec()`

```
let p = /([0-9]{2}):([0-9]{2}):([0-9]{2})/g;
let str = '18:47:27 05:12:22';
console.log( p.exec(str) );
/* [
  '18:47:27',
  '18',
  '47',
  '27',
  index: 0,
  input: '18:47:27 05:12:22',
  groups: undefined
] */
console.log( p.exec(str) );
/* [
  '05:12:22',
  '05',
  '12',
  '22',
  index: 9,
  input: '18:47:27 05:12:22',
  groups: undefined
] */
console.log( p.exec(str) ); // null
console.log( p.exec('200') ); // null
```

Первый элемент массива содержит найденную подстроку, второй, третий и четвертый элементы — фрагменты, которые соответствуют группам метасимволов `([0-9]{2})`, заключенным в круглые скобки. Номер скобок по порядку следования в регулярном выражении соответствует индексу фрагмента в массиве.

Перебрать все совпадения с помощью цикла `while` можно так:

```
let p = /([0-9]{2}):([0-9]{2}):([0-9]{2})/g;
let str = '18:47:27 05:12:22';
let arr;
while ( (arr = p.exec(str)) ) {
  console.log( arr );
}
```

Если в регулярном выражении указан модификатор `d`, то возвращенный методом массив будет дополнительно включать свойство `indices`. Оно будет хранить массив, каждый элемент которого представляет собой массив из двух элементов — индексов начального и конечного символов очередной найденной под-

строки или ее фрагмента в текущей строке. Пример использования модификатора `d` в методе `exec()` приведен в листинге 8.4.

Листинг 8.4. Использование модификатора `d` в методе `exec()`

```
let p = /([0-9]{2}):([0-9]{2}):([0-9]{2})/d;
let str = '18:47:27 05:12:22';
console.log( p.exec(str) );
// [
//   '18:47:27',
//   '18',
//   '47',
//   '27',
//   index: 0,
//   input: '18:47:27 05:12:22',
//   groups: undefined,
//   indices: [ [ 0, 8 ], [ 0, 2 ], [ 3, 5 ], [ 6, 8 ],
//   groups: undefined ]
// ]
```

Если необходимо удостовериться, что строка полностью соответствует регулярному выражению, нужно в его шаблоне указать привязку к началу (^) и концу (\$) строки. В качестве примера проверим правильность даты, введенной при запуске программы в командной строке (листинг 8.5).

Листинг 8.5. Проверка правильности введенной даты

```
if (process.argv.length == 3) {
  let d = process.argv[2];
  let p = /^[0-3][0-9]\.[01][0-9]\.[0-9]{4}$/;
  if ( p.test(d) ) {
    console.log('Дата введена правильно');
  }
  else {
    console.log('Вы неправильно ввели дату');
  }
}
else {
  console.log('Передайте дату в формате день.месяц.год в конце команды');
}
```

8.4. Свойства класса *RegExp*

Объект регулярного выражения содержит следующие свойства:

□ `lastIndex` — позиция начала поиска:

```
let p = /[0-9]+/g;
let str = '1 2 3';
```

```
console.log( p.lastIndex ); // 0
console.log( p.exec(str) );
// [ '1', index: 0, input: '1 2 3', groups: undefined ]
console.log( p.lastIndex ); // 1
console.log( p.exec(str) );
// [ '2', index: 2, input: '1 2 3', groups: undefined ]
console.log( p.lastIndex ); // 3
```

Это единственное свойство, доступное и для чтения, и для записи. Остальные свойства доступны только для чтения:

- ❑ **global** — значение `true`, если установлен модификатор `g`, и `false` — в противном случае:

```
let p = /[0-9]+/g;
console.log( p.global ); // true
console.log( /[0-9]+/.global ); // false
```

- ❑ **multiline** — значение `true`, если установлен модификатор `m`, и `false` — в противном случае:

```
let p = /^[0-9]+$/m;
console.log( p.multiline ); // true
console.log( /^[0-9]+$/multiline ); // false
```

- ❑ **ignoreCase** — значение `true`, если установлен модификатор `i`, и `false` — в противном случае:

```
let p = /[a-z]+/i;
console.log( p.ignoreCase ); // true
console.log( /[a-z]+/.ignoreCase ); // false
```

- ❑ **dotAll** — значение `true`, если установлен модификатор `s`, и `false` — в противном случае:

```
let p = ./.;
console.log( p.dotAll ); // false
p = ./s;
console.log( p.dotAll ); // true
```

- ❑ **hasIndices** — значение `true`, если установлен модификатор `d`, и `false` — в противном случае:

```
let p = ./.;
console.log( p.hasIndices ); // false
p = ./d;
console.log( p.hasIndices ); // true
```

- ❑ **sticky** — значение `true`, если установлен модификатор `y`, и `false` — в противном случае:

```
let p = ./.;
console.log( p.sticky ); // false
p = ./y;
console.log( p.sticky ); // true
```

- ❑ `unicode` — значение `true`, если установлен модификатор `u`, и `false` — в противном случае:

```
let p = /\p{Letter}+/gu;
console.log( p.unicode );    // true
```

- ❑ `source` — регулярное выражение в виде строки без модификаторов:

```
let p = /[0-9]+/g;
console.log(p.source);      // [0-9]+
p = new RegExp();
console.log(p.source);      // (?)
p = new RegExp('\\n');
console.log(p.source);      // \n
p = new RegExp('\n');
console.log(p.source);      // \n
```

- ❑ `flags` — модификаторы регулярного выражения в виде строки:

```
let p = /[a-z]+/img;
console.log(p.flags);      // gim
```

8.5. Синтаксис регулярных выражений

Обычные символы, не имеющие специального значения, могут присутствовать в шаблоне, и они будут трактоваться как есть. Пример указания в шаблоне последовательности обычных символов:

```
let p = /строка/;
console.log( p.test('строка') ); // true
```

8.5.1. Экранирование специальных символов

Внутри регулярного выражения символы `.`, `^`, `$`, `*`, `+`, `?`, `{`, `[`, `]`, `\`, `|`, `(` и `)` имеют специальное значение. Если эти символы должны трактоваться как есть, то их следует экранировать с помощью слеша. Некоторые специальные символы теряют свое специальное значение, если их разместить внутри квадратных скобок. В этом случае экранировать их не нужно. Например, метасимвол `точка` по умолчанию соответствует любому символу, кроме символа перевода строки. Если необходимо найти именно точку, то перед точкой необходимо указать символ `\` или поместить точку внутри квадратных скобок (`[.]`). Продемонстрируем это на примере проверки правильности введенной даты (листинг 8.6).

Листинг 8.6. Проверка правильности введенной даты

```
let d = '24,10.2020';
// Неправильная дата (вместо точки указана запятая)
let p = /^[0-3][0-9].[01][0-9].[12][09][0-9][0-9]$/;
```

```
// Символ \ не указан перед точкой
if ( p.test(d) ) console.log('Дата введена правильно');
else console.log('Дата введена неправильно');
// Выведет Дата введена правильно, т. к. точка означает любой символ

p = /^[0-3][0-9]\.[01][0-9]\.[12][09][0-9][0-9]$/;
// Символ \ указан перед точкой
if ( p.test(d) ) console.log('Дата введена правильно');
else console.log('Дата введена неправильно');
// Выведет Дата введена неправильно,
// т. к. перед точкой указан символ \, а в дате присутствует запятая

p = /^[0-3][0-9][.][01][0-9][.][12][09][0-9][0-9]$/;
// Точка внутри квадратных скобок
if ( p.test(d) ) console.log('Дата введена правильно');
else console.log('Дата введена неправильно');
// Выведет Дата введена неправильно

p = new RegExp('^[0-3]\\d\\. [01]\\d\\. [12][09]\\d\\d$');
// Символ \ указан перед точкой
if ( p.test(d) ) console.log('Дата введена правильно');
else console.log('Дата введена неправильно');
// Выведет Дата введена неправильно
```

Следует учитывать, что символ обратного слеша является специальным не только в шаблоне регулярного выражения, но и в строке. Поэтому его нужно экранировать при создании шаблона с помощью класса `RegExp`. Обратите внимание на регулярное выражение в последнем примере:

```
'^[0-3]\\d\\. [01]\\d\\. [12][09]\\d\\d$'
```

Внутри шаблона регулярного выражения комбинация символов `\d` является специальной и обозначает любые цифры. В строке символ `\` должен заменяться на `\\`. Поэтому вместо `\d` указано `\\d`, а вместо `\.` — `\\.`

В неформатированных строках (см. *разд. 7.4*) экранировать специальные символы не нужно:

```
let d = '24,10.2020';
let p = new RegExp( String.raw`^[0-3]\d\.[01]\d\.[12][09]\d\d$` );
if ( p.test(d) ) console.log('Дата введена правильно');
else console.log('Дата введена неправильно');
// Выведет: Дата введена неправильно
```

Метасимвол *точка* теряет свое специальное значение, если его заключить в квадратные скобки. Кроме того, внутри квадратных скобок могут присутствовать другие символы, имеющие специальное значение (например, `^` и `-`). Символ `^` теряет свое специальное значение, если он не расположен сразу после открывающей квадратной скобки:


```
let p = /[09^]/g;           // 0, 9 или ^
let str = '0123456789^';
console.log( str.match(p) ); // [ '0', '9', '^' ]
```

Чтобы отменить специальное значение символа `^`, его необходимо указать после перечисления всех символов, перед закрывающей квадратной скобкой:

```
let p = /[09-]/g;           // 0, 9 или -
let str = '0123456789-';
console.log( str.match(p) ); // [ '0', '9', '-' ]
```

Любой специальный символ можно сделать обычным, указав перед ним символ `\`:

```
let p = /[0\-9]/g;          // 0, - или 9
let str = '0123456789-';
console.log( str.match(p) ); // [ '0', '9', '-' ]
```

Символ `/` в регулярном выражении, ограниченном символами `/`, также является специальным, поэтому его нужно экранировать:

```
let p = /[0\/9]/g;          // 0, / или 9
let str = '0123456789-';
console.log( str.match(p) ); // [ '0', '9', '/' ]
```

8.5.2. Метасимволы

Метасимволы — это специальные символы, управляющие тем, как производится проверка строк. Перечислим метасимволы, применяемые в регулярных выражениях:

- ❑ `^` — привязка к началу строки. Если указан модификатор `m`, то соответствует началу каждой подстроки;
- ❑ `$` — привязка к концу строки. Если указан модификатор `m`, то соответствует концу каждой подстроки;
- ❑ `[]` — позволяет указать символы, которые могут встречаться на этом месте в строке. Можно перечислять символы подряд или указать диапазон через дефис;
- ❑ `[^]` — позволяет указать символы, которые не могут встречаться на этом месте в строке. Можно перечислять символы подряд или указать диапазон через дефис;
- ❑ `n|m` — соответствует одному из фрагментов `n` или `m`:

```
let p = /красн(ая|ое)/g;
let str = 'красная или красное, но не красный';
console.log( str.match(p) );
// [ 'красная', 'красное' ]
```

- ❑ `.` (точка) — любой символ, кроме перевода строки (`\n`). Если указан модификатор `s`, то точка соответствует любому символу, включая перевод строки. Внутри квадратных скобок точка не имеет специального значения. Пример:

```
let p = /.+/g;
let str = '10\n20';
```

```
console.log( str.match(p) ); // [ '10', '20' ]
p = /.+/gs;
console.log( str.match(p) ); // [ '10\n20' ]
p = /[.]+/gs;
console.log( str.match(p) ); // null
```

Если нужно найти соответствие любому символу, включая символ перевода строки, то можно воспользоваться следующим кодом: `[\s\S]`. Класс `\s` означает любой пробельный символ, а класс `\S` — любой непробельный символ. Пример:

```
let p = /.+/g;
let str = '10\n20';
console.log( str.match(p) ); // [ '10', '20' ]
p = /[\s\S]+/g;
console.log( str.match(p) ); // [ '10\n20' ]
```

Привязку к началу и концу строки следует использовать, если строка должна полностью соответствовать регулярному выражению. Например, проверим, содержит ли строка только число:

```
let p = /^[0-9]+$/; // Строка может содержать только числа
if ( p.test('2') ) console.log('Число'); // Выведет: Число
else console.log('Не число');
if ( p.test('Строка2') ) console.log('Число');
else console.log('Не число'); // Выведет: Не число
```

Не указав привязку к началу и концу, мы получим соответствие шаблону любого числа внутри строки:

```
let p = /[0-9]+/;
if ( p.test('Строка2') )
    console.log('Есть число'); // Выведет: Есть число
else console.log('Нет числа');
```

Можно указать привязку только к началу или только к концу строки:

```
let p = /[0-9]+$/;
if ( p.test('Строка2') ) console.log('Есть число в конце строки');
else console.log('Нет числа в конце строки');
// Выведет: Есть число в конце строки
p = /^[0-9]+/;
if ( p.test('Строка2') ) console.log('Есть число в начале строки');
else console.log('Нет числа в начале строки');
// Выведет: Нет числа в начале строки
```

В квадратных скобках `[]` указываются символы, которые могут встречаться на этом месте в строке. Можно перечислить их подряд или указать диапазон через дефис. Примеры:

- ❑ `[09]` — соответствует цифре 0 или 9;
- ❑ `[0-9]` — соответствует любой цифре от 0 до 9;

- ❑ [абв] — соответствует буквам "а", "б" и "в";
- ❑ [а-г] — соответствует буквам "а", "б", "в" и "г";
- ❑ [а-яё] — соответствует любой букве от "а" до "я";
- ❑ [АВВ] — соответствует буквам "А", "Б" и "В". Обратите внимание, если не указан модификатор *i*, регистр будет иметь значение;
- ❑ [А-ЯЁ] — соответствует любой букве от "А" до "Я";
- ❑ [а-яА-ЯёЁ] — соответствует любой русской букве в любом регистре;
- ❑ [0-9а-яА-ЯёЁа-зА-З] — любая цифра и любая кириллическая или латинская буква независимо от регистра.

ОБРАТИТЕ ВНИМАНИЕ

Буква "ё" не входит в диапазон [а-я].

Значение можно инвертировать, если после первой скобки указать символ `^`. Таким образом можно указать символы, которых не должно быть на этом месте в строке.

Примеры:

- ❑ [^09] — не цифра 0 или 9;
- ❑ [^0-9] — не цифра от 0 до 9;
- ❑ [^а-яА-ЯёЁа-зА-З] — не кириллическая или латинская буква в любом регистре.

8.5.3. Стандартные классы

В регулярных выражениях допустимы следующие основные *стандартные классы* (полный список приведен в документации):

- ❑ `\d` — соответствует любой цифре;
- ❑ `\w` — соответствует любой латинской букве, цифре и знаку подчеркивания (`[a-zA-Z0-9_]`);
- ❑ `\s` — любой пробельный символ (пробел, табуляция, перевод страницы, новая строка или перевод каретки);
- ❑ `\b` — граница слова;
- ❑ `\D` — не цифра. Эквивалентно: `[^\d]`;
- ❑ `\W` — не латинская буква, не цифра и не знак подчеркивания. Эквивалентно: `[^\w]`;
- ❑ `\S` — непробельный символ. Эквивалентно: `[^\s]`.

ВНИМАНИЕ!

Метасимвол `\w` работает только с буквами латинского алфавита. С буквами русского языка он не работает.

Следует учитывать, что стандартные классы могут зависеть от различных флагов и трактоваться гораздо шире, чем мы указали. В итоге можно получить результат,

который совсем не ожидался. Советуем на практике использовать только классы `\s` и `\S`, а остальные заменять явным указанием диапазона символов внутри квадратных скобок (при этом не забывайте о букве "ё"):

```
let p = /[0-9]/g;           // Вместо \d
let str = '10 23 45';
console.log( str.match(p) ); // [ '1', '0', '2', '3', '4', '5' ]
p = /[a-zA-Za-яА-ЯёЁ0-9_]/g; // Вместо \w
str = 'Z ш Ё _ 0';
console.log( str.match(p) ); // [ 'Z', 'ш', 'Ё', '_', '0' ]
```

Если нужно получить не цифры по отдельности, а числа, то после закрывающей квадратной скобки следует указать символ `+`, означающий одно или большее число вхождений символа в строку:

```
let p = /[0-9]+/g;
let str = '10 23 45';
console.log( str.match(p) ); // [ '10', '23', '45' ]
```

8.5.4. Квантификаторы

Число вхождений символа в строку задается с помощью *квантификаторов*:

- `{n}` — n вхождений символа в строку (шаблон `[0-9]{2}` соответствует двум вхождениям любой цифры);
- `{n,}` — n или более вхождений символа в строку (шаблон `[0-9]{2,}` соответствует двум и более вхождениям любой цифры);
- `{n,m}` — не менее n и не более m вхождений символа в строку. Числа указываются через запятую без пробела. Например, шаблон `[0-9]{2,4}` соответствует двум, трем или четырем вхождениям любой цифры;
- `*` — ноль или большее число вхождений символа в строку. Эквивалентно комбинации `{0,}`;
- `+` — одно или большее число вхождений символа в строку. Эквивалентно комбинации `{1,}`;
- `?` — ни одного или одно вхождение символа в строку. Эквивалентно комбинации `{0,1}`.

8.5.5. "Жадность" квантификаторов

Все квантификаторы являются "жадными". При поиске соответствия ищется самая длинная подстрока, соответствующая шаблону, и не учитываются более короткие соответствия. Рассмотрим это на примере. Получим все теги `<b>` вместе с их содержанием:

```
let p = /<b>.*</b>/gi;
let str = '<b>Text1</b>Text2<b>Text3</b>';
console.log( str.match(p) );
// [ '<b>Text1</b>Text2<b>Text3</b>' ]
```

Вместо желаемого результата мы получили полностью строку. Чтобы ограничить "жадность", необходимо после квантификатора указать символ ?. Пример:

```
let p = /<b>.*?</b>/gi;
let str = '<b>Text1</b>Text2<b>Text3</b>';
console.log( str.match(p) );
// [ '<b>Text1</b>', '<b>Text3</b>' ]
```

8.5.6. Группы

Если необходимо получить содержимое без тегов, то нужный фрагмент внутри шаблона следует разместить внутри круглых скобок:

```
let p = /<b>(.*?)</b>/gi;
let str = '<b>Text1</b>Text2<b>Text3</b>';
let arr;
while ( (arr = p.exec(str)) ) {
  console.log(arr);
}
/* [
  '<b>Text1</b>',
  'Text1',
  index: 0,
  input: '<b>Text1</b>Text2<b>Text3</b>',
  groups: undefined
]
[
  '<b>Text3</b>',
  'Text3',
  index: 17,
  input: '<b>Text1</b>Text2<b>Text3</b>',
  groups: undefined
] */
```

Результат будет доступен через элемент массива с индексом, совпадающим с порядковым номером круглых скобок внутри шаблона (нумерация начинается с 1). Элемент массива с индексом 0 будет содержать саму подстроку, совпавшую с регулярным выражением.

Круглые скобки часто используются для группировки фрагментов внутри шаблона. В этих случаях не требуется, чтобы фрагмент запоминался и был доступен в результатах поиска. Пример использования круглых скобок приведен в листинге 8.7.

Листинг 8.7. Применение круглых скобок для группировки фрагментов

```
let p = /[a-z]+((st)|(xt))/gi;
let str = 'test text';
let arr;
```

```

while ( (arr = p.exec(str)) ) {
    console.log(arr);
}
/* [
    'test',
    'st',
    'st',
    undefined,
    index: 0,
    input: 'test text',
    groups: undefined
]
[
    'text',
    'xt',
    undefined,
    'xt',
    index: 5,
    input: 'test text',
    groups: undefined
] */

```

Три последних элемента в этом примере являются лишними. Чтобы избежать захвата фрагмента, после открывающей круглой скобки следует разместить символы `?:`. Пример их использования приведен в листинге 8.8.

Листинг 8.8. Применение символов `?:` для предотвращения захвата фрагмента

```

let p = /[a-z]+(?: (?:st) | (?:xt)) /gi;
let str = 'test text';
let arr;
while ( (arr = p.exec(str)) ) {
    console.log(arr);
}
/*
[ 'test', index: 0, input: 'test text', groups: undefined ]
[ 'text', index: 5, input: 'test text', groups: undefined ]
*/

```

В качестве примера использования групп разберем адрес электронной почты (листинг 8.9) и URL-адрес (листинг 8.10) на составные части.

Листинг 8.9. Разбор адреса электронной почты на составные части

```

let p = /^([a-z0-9_-.]+)@((?:[a-z0-9-]+\.)+[a-z]{2,6})$/i;
let str = 'user@mail.ru';
let arr = p.exec(str);

```

```

if (arr) {
    console.log(arr);
    console.log('Имя ящика - ' + arr[1]);
    console.log('Имя сайта - ' + arr[2]);
    console.log('Полный E-mail - ' + arr[0]);
}
else {
    console.log('E-mail не соответствует шаблону');
}

```

В итоге получим следующий результат:

```

[
  'user@mail.ru',
  'user',
  'mail.ru',
  index: 0,
  input: 'user@mail.ru',
  groups: undefined
]
Имя ящика - user
Имя сайта - mail.ru
Полный E-mail - user@mail.ru

```

Листинг 8.10. Разбор интернет-адреса на составные части

```

let s = '^([a-z]+:\/) ' +
    '((?:[a-z0-9-]+\.\.)+[a-z]{2,6}) ' +
    '([a-z0-9-\/]*\/)* ' +
    '([a-z0-9-]+\.\.[a-z]+)$';
let p = new RegExp(s, 'i');
let str = 'http://www.mysite.ru/folder1/folder2/forder3/file.html';
let arr = p.exec(str);
if (arr) {
    console.log('Протокол - ' + arr[1]);
    console.log('Сайт - ' + arr[2]);
    console.log('Путь - ' + arr[3]);
    console.log('Имя файла - ' + arr[4]);
}
else {
    console.log('URL не соответствует шаблону');
}

```

В итоге получим следующий результат:

```

Протокол - http://
Сайт - www.mysite.ru
Путь - /folder1/folder2/forder3/
Имя файла - file.html

```

Для получения фрагментов можно также воспользоваться статическими свойствами `$1 ... $N`, а для получения полного соответствия — свойством `lastMatch`:

```
console.log('Полный URL - ' + RegExp.lastMatch);
console.log('Протокол - ' + RegExp.$1);
console.log('Сайт - ' + RegExp.$2);
console.log('Путь - ' + RegExp.$3);
console.log('Имя файла - ' + RegExp.$4);
```

8.5.7. Обратные ссылки

К найденному фрагменту в круглых скобках внутри шаблона можно обратиться с помощью *обратных ссылок*. Для этого порядковый номер круглых скобок в шаблоне указывается после слеша, например `\1`. Нумерация скобок внутри шаблона начинается с 1. Для примера получим текст между одинаковыми парными тегами (см. листинг 8.11).

Листинг 8.11. Получение содержимого парных тегов

```
let p = /<([a-z]+) [^>]*?>([\s\S]*?)<\/\1>/gi;
let str = '<b>Text1</b>Text2<I>Text3</I>';
let arr;
while ( (arr = p.exec(str)) ) {
  console.log(arr);
}
/* [
  '<b>Text1</b>',
  'b',
  'Text1',
  index: 0,
  input: '<b>Text1</b>Text2<I>Text3</I>',
  groups: undefined
]
[
  '<I>Text3</I>',
  'I',
  'Text3',
  index: 17,
  input: '<b>Text1</b>Text2<I>Text3</I>',
  groups: undefined
] */
```

8.5.8. Просмотр вперед и назад

Также поддерживаются следующие конструкции:

- `(?=...)` — положительный просмотр вперед. Выведем все слова, после которых расположена запятая:


```
let p = /\w+(?=[,])/g;
let str = 'text1, text2, text3 text4';
console.log( str.match(p) ); // [ 'text1', 'text2' ]
```

- ❑ **(?!...)** — отрицательный просмотр вперед. Выведем все слова, после которых нет запятой:

```
let p = /[a-z]+[0-9](?![,])/g;
let str = 'text1, text2, text3 text4';
console.log( str.match(p) ); // [ 'text3', 'text4' ]
```

- ❑ **(?<=...)** — положительный просмотр назад. Выведем все числа, перед которыми стоит минус:

```
let p =/(?<=)\d+/g;
let str = '100 -768 5206759 -4589';
console.log( str.match(p) ); // [ '768', '4589' ]
```

- ❑ **(?<!...)** — отрицательный просмотр назад. Выведем все числа, перед которыми нет минуса:

```
let p = /\b(?<!\-)\d+\b/g;
let str = '100 -768 5206759 -4589';
console.log( str.match(p) ); // [ '100', '5206759' ]
```

Рассмотрим небольшой пример. Предположим, что необходимо получить все слова, расположенные после тире, причем перед тире и после слов должны следовать пробельные символы. Для этого мы напишем такой код:

```
let p = /\s\-([a-z0-9]+)\s/g;
let str = '-word1 -word2 -word3 -word4 -word5';
let arr;
while ( (arr = p.exec(str)) ) {
  process.stdout.write(arr[1] + ' ');
}
// word2 word4
```

Мы получили только два слова вместо пяти. Первое и последнее слова не попали в результат, т. к. расположены в начале и в конце строки. Чтобы эти слова попали в результат, необходимо добавить альтернативные выборы: **(^|\s)** — для начала строки и **(\s|\$)** — для конца строки. Чтобы найденные выражения внутри круглых скобок не попали в результат, следует добавить символы **?:** после открывающей скобки. Исправленный код:

```
let p = /(?:^|\s)\-([a-z0-9]+)(?:\s|$)/g;
let str = '-word1 -word2 -word3 -word4 -word5';
let arr;
while ( (arr = p.exec(str)) ) {
  process.stdout.write(arr[1] + ' ');
}
// word1 word3 word5
```

Первое и последнее слова успешно попали в результат. Почему же слова `word2` и `word4` не попали в список совпадений? Ведь и перед тире есть пробел, и после слова есть пробел. Чтобы понять причину, рассмотрим поиск по шагам. Первое слово успешно попадает в результат, т. к. перед тире расположено начало строки и после слова есть пробел. После поиска указатель перемещается, и строка для дальнейшего поиска примет следующий вид:

```
"-word1 <Указатель>-word2 -word3 -word4 -word5"
```

Обратите внимание на то, что перед фрагментом `-word2` больше нет пробела и тире не расположено в начале строки. Поэтому следующим совпадением станет слово `word3` и указатель снова будет перемещен:

```
"-word1 -word2 -word3 <Указатель>-word4 -word5"
```

Опять перед фрагментом `-word4` нет пробела и тире не расположено в начале строки. Поэтому следующим совпадением станет слово `word5` и поиск будет завершен. Таким образом, слова `word2` и `word4` не попадают в результат, поскольку пробел до фрагмента уже был использован в предыдущем поиске. Чтобы этого избежать, следует воспользоваться положительным просмотром вперед (`?=...`), исправив код следующим образом:

```
let p = /(?:^|\s)\-([a-z0-9]+)(?=\s|$)/g;
let str = '-word1 -word2 -word3 -word4 -word5';
let arr;
while ( (arr = p.exec(str)) ) {
    process.stdout.write(arr[1] + ' ');
}
// word1 word2 word3 word4 word5
```

В этом примере мы заменили фрагмент `(?:\s|$)` на `(?=\s|$)`. Поэтому все слова успешно попали в список совпадений.

Другой вариант — совместное использование положительного просмотра вперед и назад:

```
let p = /(?!<=^|\s)\-([a-z0-9]+)(?=\s|$)/g;
```

Результат, выдаваемый таким регулярным выражением, будет аналогичен приведенному ранее.



ГЛАВА 9

Работа с датой и временем

Работать с датой и временем позволяет класс `Date`. Экземпляры класса создаются так:

```
<Экземпляр класса> = new Date()  
<Экземпляр класса> = new Date(<Число миллисекунд>)  
<Экземпляр класса> = new Date(<Год>, <Месяц>[, <День>[, <Часы>[,  
                                <Минуты>[, <Секунды>[, <Миллисекунды>]]]]])  
<Экземпляр класса> = new Date(<Строка с датой>)
```

9.1. Получение текущей даты и времени

При использовании первого формата возвращается объект с текущей датой и временем:

```
let d = new Date();  
console.log(d);  
// Node.js 16: 2021-04-26T15:31:48.048Z  
// Firefox: Mon Apr 26 2021 18:31:48 GMT+0300 (Москва, стандартное время)  
console.log( d.toString() );  
console.log( d.toLocaleString() );  
// 26.04.2021, 18:31:48
```

Получить число миллисекунд, прошедшее с 1 января 1970 г., позволяет статический метод `now()`:

```
let t = Date.now();  
console.log(t); // 1619451291865
```

9.2. Указание произвольных значений даты и времени

При использовании второго формата дата будет соответствовать числу миллисекунд, прошедших с 1 января 1970 г.:

```
let d = new Date(1619451291865);
console.log( d.toLocaleString() );
// 26.04.2021, 18:34:51
```

Указать число миллисекунд, прошедшее с 1 января 1970 г., после создания объекта позволяет метод `setTime()`:

```
let d = new Date();
d.setTime(1619451291865);
console.log( d.toLocaleString() );
// 26.04.2021, 18:34:51
```

При использовании третьего формата можно задать произвольные значения:

```
let d = new Date(2021, 3, 26, 18, 37, 46);
console.log( d.toLocaleString() );
// 26.04.2021, 18:37:46
```

Значения задаются в виде чисел. Кроме того, в параметре `<Месяц>` указываются числа от 0 (январь) до 11 (декабрь), а не от 1 до 12. Если указать число 12, то месяц станет январем, а к году будет прибавлена единица.

Получить число миллисекунд на основе произвольных значений даты и времени позволяет статический метод `UTC()`:

```
<Время> = Date.UTC(<Год>[, <Месяц>[, <День>[, <Часы>[,
                    <Минуты>[, <Секунды>[, <Миллисекунды>]]]]]]);
```

Пример:

```
let t = Date.UTC(2021, 3, 26, 18, 37, 46);
console.log(t); // 1619462266000
console.log( ( new Date(t) ).toString() );
// Mon Apr 26 2021 21:37:46 GMT+0300 (Москва, стандартное время)
```

9.3. Разбор строки с датой и временем

Четвертый формат предназначен для разбора строки с датой и временем:

```
let d = new Date('Mon Apr 26 2021 21:37:46 GMT+0300');
console.log( d.toLocaleString() );
// 26.04.2021, 21:37:46
```

Если разобрать строку не удалось, возвращается объект `Date` со значением `Invalid Date`:

```
let d = new Date('Mon Apr 21:37:46 GMT+0300');
console.log( d ); // Invalid Date
console.log( typeof d ); // object
console.log( d instanceof Date ); // true
console.log( d.getTime() ); // NaN
```

Получить число миллисекунд, прошедшее с 1 января 1970 г., на основе строки с датой и временем, позволяет статический метод `parse()`:

```
let t = Date.parse('Mon Apr 26 2021 21:37:46 GMT+0300');
console.log(t); // 1619462266000
console.log( ( new Date(t) ).toString() );
// Mon Apr 26 2021 21:37:46 GMT+0300 (Москва, стандартное время)
```

Если разобрать строку не удалось, метод `parse()` вернет значение `NaN`:

```
let t = Date.parse('Mon Apr 21:37:46 GMT+0300');
console.log(t); // NaN
```

9.4. Преобразование даты в строку

Преобразовать значение объекта класса `Date` в строку позволяют следующие методы:

- ❑ `toString()` — преобразует дату в строку и возвращает ее:

```
let d = new Date(1619451291865);
console.log( d.toString() );
// Mon Apr 26 2021 18:34:51 GMT+0300 (Москва, стандартное время)
```

- ❑ `toLocaleString([<Локаль>[, <Опции>]])` — преобразует дату в строку, используя настройки локали:

```
let d = new Date(1619451291865);
console.log( d.toLocaleString() );
// 26.04.2021, 18:34:51
```

- ❑ `toUTCString()` — преобразует дату в строку с учетом часового пояса UTC:

```
let d = new Date(1619451291865);
console.log( d.toUTCString() );
// Mon, 26 Apr 2021 15:34:51 GMT
```

- ❑ `toISOString()` и `toJSON()` — преобразует дату в строку в формате ISO:

```
let d = new Date(1619451291865);
console.log( d.toISOString() );
// 2021-04-26T15:34:51.865Z
console.log( d.toJSON() );
// 2021-04-26T15:34:51.865Z
```

- ❑ `toDateString()` — возвращает строковое представление даты:

```
let d = new Date(1619451291865);
console.log( d.toDateString() );
// Mon Apr 26 2021
```

- ❑ `toLocaleDateString([<Локаль>[, <Опции>]])` — возвращает строковое представление даты с учетом настроек локали:

```
let d = new Date(1619451291865);
console.log( d.toLocaleDateString() );
// 26.04.2021
```

- ❑ `toString()` — возвращает строковое представление времени:

```
let d = new Date(1619451291865);
console.log( d.toString() );
// 18:34:51 GMT+0300 (Москва, стандартное время)
```

- ❑ `toLocaleTimeString([<Локаль>[, <Опции>]])` — возвращает строковое представление времени с учетом настроек локали:

```
let d = new Date(1619451291865);
console.log( d.toLocaleTimeString() );
// 18:34:51
```

Альтернативный способ получения строкового представления даты:

```
let d = new Date(1619451291865);
let str = d[Symbol.toPrimitive]('string');
console.log(str);
// Mon Apr 26 2021 18:34:51 GMT+0300 (Москва, стандартное время)
str = d[Symbol.toPrimitive]('default');
console.log(str);
// Mon Apr 26 2021 18:34:51 GMT+0300 (Москва, стандартное время)
```

9.5. Получение и изменение значений компонентов даты и времени

Получить значения отдельных компонентов даты и времени позволяют методы из класса `Date`, перечисленные далее. Методы, содержащие в названии фрагмент UTC, возвращают значения в часовом поясе UTC, а без него — по местному времени.

- ❑ `valueOf()` и `getTime()` — возвращают число миллисекунд, прошедшее с 1 января 1970 г.:

```
let d = new Date(1619451291865);
console.log( d.valueOf() ); // 1619451291865
console.log( d.getTime() ); // 1619451291865
```

Альтернативный способ получения значения:

```
let d = new Date(1619451291865);
console.log( d[Symbol.toPrimitive]('number') );
// 1619451291865
```

- ❑ `getDate()` и `getUTCDate()` — возвращают день месяца (от 1 до 31):

```
let d = new Date(1619451291865);
console.log( d.getDate() ); // 26
console.log( d.getUTCDate() ); // 26
```

- ❑ `getMonth()` и `getUTCMonth()` — возвращают номер месяца (от 0 — для января до 11 — для декабря):

```
let months = [ 'январь', 'февраль', 'март', 'апрель', 'май',  
               'июнь', 'июль', 'август', 'сентябрь', 'октябрь',  
               'ноябрь', 'декабрь' ];  
let d = new Date(1619451291865);  
console.log( d.getMonth() );           // 3  
console.log( d.getUTCMonth() );         // 3  
console.log( months[ d.getMonth() ] ); // апрель
```

Для получения номера текущего месяца к возвращаемому значению необходимо прибавить единицу:

```
let d = new Date(1619451291865);  
console.log( d.getMonth() + 1 );       // 4
```

- ❑ `getFullYear()` и `getUTCFullYear()` — возвращают год из четырех цифр:

```
let d = new Date(1619451291865);  
console.log( d.getFullYear() );        // 2021  
console.log( d.getUTCFullYear() );      // 2021
```

- ❑ `getDay()` и `getUTCDay()` — возвращают номер дня недели (от 0 — для воскресенья до 6 — для субботы):

```
let days = [ 'воскресенье', 'понедельник', 'вторник',  
             'среда', 'четверг', 'пятница', 'суббота' ];  
let d = new Date(1619451291865);  
console.log( d.getDay() );             // 1  
console.log( d.getUTCDay() );          // 1  
console.log( days[ d.getDay() ] );     // понедельник
```

- ❑ `getHours()` и `getUTCHours()` — возвращают часы (от 0 до 23):

```
let d = new Date(1619451291865);  
console.log( d.getHours() );           // 18  
console.log( d.getUTCHours() );        // 15
```

- ❑ `getMinutes()` и `getUTCMinutes()` — возвращают минуты (от 0 до 59):

```
let d = new Date(1619451291865);  
console.log( d.getMinutes() );         // 34  
console.log( d.getUTCMinutes() );      // 34
```

- ❑ `getSeconds()` и `getUTCSeconds()` — возвращают секунды (от 0 до 59):

```
let d = new Date(1619451291865);  
console.log( d.getSeconds() );         // 51  
console.log( d.getUTCSeconds() );      // 51
```

- ❑ `getMilliseconds()` и `getUTCMilliseconds()` — возвращают миллисекунды (от 0 до 999):

```
let d = new Date(1619451291865);  
console.log( d.getMilliseconds() );    // 865  
console.log( d.getUTCMilliseconds() ); // 865
```

❑ `getTimezoneOffset()` — возвращает смещение зоны местного времени в минутах:

```
let d = new Date(1619451291865);
console.log( d.getTimezoneOffset() ); // -180
```

Чтобы задать значения компонентов даты и времени, нужно заменить в названиях методов префикс `get` префиксом `set` (исключением является метод `getTimezoneOffset()`):

```
let d = new Date();
d.setDate(18);
d.setMonth(3);
d.setFullYear(2021);
d.setHours(19);
d.setMinutes(3);
d.setSeconds(21);
d.setMilliseconds(185);
console.log( d.toString() );
// Sun Apr 18 2021 19:03:21 GMT+0300 (Москва, стандартное время)
```

9.6. Форматированный вывод текущей даты и времени

Рассмотрим работу с датой и временем на примере. Выведем текущую дату и время в консоль (листинг 9.1).

Листинг 9.1. Вывод текущей даты и времени

```
function numToStr(value) {
    value += ''; // Преобразуем число в строку
    if (value.length == 1) return '0' + value;
    else return value;
}

function yearToStr(value) {
    value += ''; // Преобразуем число в строку
    return value.substring(2);
}

let d = new Date();
let days = [ 'воскресенье', 'понедельник', 'вторник', 'среда',
             'четверг', 'пятница', 'суббота' ];
let months = [ 'января', 'февраля', 'марта', 'апреля', 'мая',
               'июня', 'июля', 'августа', 'сентября', 'октября',
               'ноября', 'декабря' ];

let msg = 'Сегодня:\n' + days[ d.getDay() ] + ' ';
msg += d.getDate() + ' ';
msg += months[ d.getMonth() ] + ' ';
msg += d.getFullYear() + ' ';
```



```

msg += numToStr( d.getHours() ) + ':';
msg += numToStr( d.getMinutes() ) + ':';
msg += numToStr( d.getSeconds() ) + '\n';
msg += numToStr( d.getDate() ) + '.';
msg += numToStr( d.getMonth() + 1 ) + '.';
msg += yearToStr( d.getFullYear() );
console.log(msg);

```

В консоли отобразится надпись (в другое время надпись будет иной, т. к. мы работаем с текущим временем):

```

Сегодня:
вторник 27 апреля 2021 10:31:07
27.04.21

```

В примере мы использовали две созданные нами функции:

- `numToStr()` — если параметр состоит из одной цифры, функция добавляет перед ним 0 и возвращает строку. Если не применить эту функцию, дата 05.04.2020 будет выглядеть как 5.4.2020;
- `yearToStr()` — возвращает последние две цифры года.

9.7. Таймеры

Таймеры позволяют однократно или многократно выполнять указанную функцию через определенный интервал времени. Для управления таймерами используются следующие глобальные функции (в веб-браузере это методы объекта `window`):

- `setTimeout()` — создает таймер, однократно выполняющий указанную функцию (или выражение в веб-браузере) спустя заданный интервал времени:

```

<Идентификатор> = setTimeout(<Функция или выражение>,
                             <Интервал>[, <Значения через запятую>])

```

Заданные третьим и последующими параметрами значения будут переданы в вызываемую функцию в качестве параметров.

В качестве результата возвращает идентификатор созданного таймера.

Для быстрой вставки функции `setTimeout()` в редакторе VS Code предусмотрена следующая комбинация:

```

setTimeout =>
setTimeout(() => {

}, timeout);

```

- `clearTimeout(<Идентификатор>)` — останавливает таймер с указанным идентификатором, созданный функцией `setTimeout()`;
- `setInterval()` — создает таймер, многократно выполняющий указанную функцию (или выражение в веб-браузере) через заданный интервал времени:

```
<Идентификатор> = setInterval(<Функция или выражение>,  
                               <Интервал>[, <Значения через запятую>])
```

Заданные третьим и последующими параметрами значения будут переданы в вызываемую функцию в качестве параметров.

В качестве результата возвращает идентификатор созданного таймера.

Для быстрой вставки функции `setInterval()` в редакторе VS Code предусмотрена следующая комбинация:

```
setinterval =>  
setInterval(() => {  
  
}, interval);
```

❑ `clearInterval(<Идентификатор>)` — останавливает таймер с указанным идентификатором, созданный функцией `setInterval()`.

Здесь `<Интервал>` — это промежуток времени, по истечении которого выполняется `<Функция или выражение>` (в Node.js возможно указание только функции). Значение задается в миллисекундах. В аргументе `<Параметры>` можно указать значения через запятую. Эти значения будут переданы в функцию в качестве параметров.

НА ЗАМЕТКУ

Не гарантируется, что функция будет вызвана сразу после истечения заданного интервала времени. Если выполняется какая-либо другая операция, функция будет вызвана после завершения этой операции. Гарантируется лишь то, что функция не будет вызвана раньше истечения установленного интервала.

Давайте однократно выполним функцию через две секунды:

```
console.log('До таймера');  
const timerId = setTimeout( function(x) {  
    console.log('Интервал истек. x =', x);  
}, 2000, 15);  
// clearTimeout(timerId);  
console.log('Конец программы');
```

Результат в консоли (последняя строка отобразится через две секунды, а не сразу):

```
До таймера  
Конец программы  
Интервал истек. x = 15
```

Если удалить символы комментария перед вызовом функции `clearTimeout()`, то результат будет таким:

```
До таймера  
Конец программы
```

Во многих языках программирования при достижении конца программы все потоки и сама программа завершаются. В Node.js это не так. После выполнения последней инструкции выполняется бесконечный цикл обработки событий, в котором ожида-

ется завершение всех выполняющихся задач. Программа завершается только в том случае, если список событий пуст.

Теперь выполним функцию три раза с интервалом в две секунды:

```
console.log('До таймера');
let count = 0;
const timerId = setInterval( function(x) {
  console.log(`count = ${count} x = ${x}`);
  count++;
  if (count > 2) clearInterval(timerId);
}, 2000, 34);
console.log('Конец программы');
```

Результат в консоли:

```
До таймера
Конец программы
count = 0 x = 34
count = 1 x = 34
count = 2 x = 34
```

В Node.js доступны две дополнительные функции:

- `setImmediate()` — создает таймер, выполняющий заданную функцию в конце текущего цикла событий:

```
<Идентификатор> = setImmediate(<Функция>[,  
                                <Значения через запятую>])
```

Заданные вторым и последующими параметрами значения будут переданы в вызываемую функцию в качестве параметров.

В качестве результата возвращает идентификатор созданного таймера;

- `clearImmediate(<Идентификатор>)` — останавливает таймер с заданным идентификатором, созданный функцией `setImmediate()`.

Пример:

```
console.log('До таймера');
const timerId = setImmediate( function(x) {
  console.log('Вызов функции. x =', x);
}, 15);
// clearInterval(timerId);
console.log('Конец программы');
```

Результат в консоли (последняя строка отобразится сразу после вывода сообщения Конец программы):

```
До таймера
Конец программы
Вызов функции. x = 15
```

Если удалить символы комментария перед вызовом функции `clearImmediate()`, то результат будет таким:

```
До таймера
Конец программы
```

9.8. Измерение времени выполнения кода

В некоторых случаях (например, при оптимизации программы) необходимо измерить время выполнения фрагментов кода. Для этого можно сохранить метку времени UNIX в переменной до начала выполнения операции, а после завершения операции повторно получить метку времени UNIX и вычесть первое время из второго.

Пример:

```
let start = Date.now();
setTimeout( function() {
    let end = Date.now();
    console.log('Интервал:', end - start);
}, 2000);
console.log('Конец программы');
```

Результат в консоли:

```
Конец программы
Интервал: 2043
```

Можно также воспользоваться следующими методами объекта `console`:

- ❑ `time([<Метка>])` — запускает таймер. В качестве параметра можно указать метку в виде строки. Если параметр не указан, то используется метка `'default'`;
- ❑ `timeLog(<Метка>[, ...<Данные>])` — выводит указанные данные в консоль, не останавливая таймер. В качестве первого параметра можно указать метку в виде строки, которая использовалась при запуске таймера. Если параметр имеет значение `undefined`, то метка будет иметь значение `'default'`;
- ❑ `timeEnd([<Метка>])` — останавливает таймер и выводит результат — время, прошедшее с момента запуска таймера, — в консоль. В качестве параметра можно указать метку в виде строки, которая использовалась при запуске таймера. Если параметр не указан, то метка будет иметь значение `'default'`.

Пример измерения времени выполнения фрагмента кода с выводом промежуточного результата:

```
console.time('myTimer');
setTimeout( function() {
    console.timeEnd('myTimer');
}, 2000);
console.log('Вывод промежуточного результата:');
console.timeLog('myTimer', 10, 20);
console.log('Конец программы');
```

Результат в консоли:

Вывод промежуточного результата:

```
myTimer: 26.464ms 10 20
```

Конец программы

```
myTimer: 2.016s
```

Пример без указания метки:

```
console.time();
```

```
setTimeout( function() {
```

```
    console.timeEnd();
```

```
}, 2000);
```

```
console.log('Вывод промежуточного результата:');
```

```
console.timeLog(undefined, 10, 20);
```

```
console.log('Конец программы');
```

Результат в консоли:

Вывод промежуточного результата:

```
default: 14.606ms 10 20
```

Конец программы

```
default: 2.010s
```



ГЛАВА 10

Пользовательские функции

Функция — это фрагмент кода JavaScript, который можно вызывать на выполнение многократно из любого места программы. В предыдущих главах мы уже не раз встречались со встроенными функциями (например, с помощью функции `parseInt()` преобразовывали строку в целое число). В этой главе мы рассмотрим создание пользовательских функций, которые позволят уменьшить избыточность программного кода и повысить его структурированность.

10.1. Создание функции и ее вызов

Функция описывается с помощью ключевого слова `function` по следующей схеме:

```
function <Имя функции>([<Параметры через запятую>]) {  
    <Тело функции>  
    [return[ <Возвращаемое значение>]];  
}
```

Имя функции должно быть допустимым идентификатором, к которому предъявляются такие же требования, как и к именам переменных. После имени функции, внутри круглых скобок, указываются имена параметров через запятую, которые являются локальными переменными, видимыми только внутри функции, создаваемыми при вызове функции, а после выхода из функции удаляемыми. Параметров может вообще не быть — в этом случае указываются только круглые скобки.

После описания параметров, внутри фигурных скобок, размещается тело функции — инструкции, выполняющиеся при каждом вызове функции. Фигурные скобки указываются в любом случае, даже если тело функции состоит только из одной инструкции. Точка с запятой после закрывающей фигурной скобки не ставится.

Функция может возвращать значение, задаваемое с помощью оператора `return`. Если оператор `return` отсутствует или указан без возвращаемого значения, функция вернет значение `undefined`.

Пример функции без параметров:

```
function showOK() {  
    console.log('Сообщение при удачно выполненной операции');  
}
```

Пример функции с одним параметром:

```
function showMessage(msg) {  
    console.log(msg);  
}
```

Пример функции с двумя параметрами, которая возвращает сумму двух переменных:

```
function sum(x, y) {  
    let z = x + y;  
    return z;  
}
```

При вызове функции указывается имя функции, после которого внутри круглых скобок через запятую записываются значения передаваемых ей параметров. Если функция не принимает параметров, указываются только круглые скобки. Если функция возвращает значение, его можно присвоить переменной или просто проигнорировать. Пример вызова функций:

```
showOK(); // Сообщение при удачно выполненной операции  
showMessage('Сообщение'); // Сообщение  
let n = sum(5, 2); // Переменной n будет присвоено значение 7  
showMessage(`n = ${n}`); // n = 7
```

В качестве возвращаемого значения в операторе `return` можно указывать не только имя переменной, но и выражение:

```
function sum(x, y) {  
    return x + y;  
}
```

Если нужно вернуть несколько значений, то значения перечисляются через запятую внутри квадратных скобок (т. е. фактически возвращается массив):

```
function func(x) {  
    return [x ** 2, x ** 3];  
}  
let [a, b] = func(5);  
console.log(a + ' ' + b); // 25 125
```

Инструкции, указанные после оператора `return`, никогда не будут выполнены:

```
function sum(x, y) {  
    return x + y;  
    console.log('Сообщение'); // Эта инструкция никогда не будет выполнена  
}
```

В консоли веб-браузера в этом случае отобразится предупреждающее сообщение "unreachable code after return statement".

Имя переменной, передающей значение функции, может не совпадать с именем соответствующего параметра:

```
function sum(x, y) {  
    return x + y;  
}  
let var1 = 5, var2 = 2, var3;  
var3 = sum(var1, var2);
```

Ссылку на функцию можно сохранить в какой-либо переменной. Для этого имя функции указывается без круглых скобок. Пример:

```
function test() {  
    console.log('Это функция test()');  
}  
let func;  
func = test; // Присваиваем ссылку на функцию  
func();      // Вызываем функцию test() через переменную func  
console.log(typeof test); // function
```

После создания функции ее можно сразу вызвать следующим способом:

```
(function test() {  
    console.log('Это функция test()');  
})();  
// Это функция test()  
// test(); // Ошибка, функция здесь не видна  
// ReferenceError: test is not defined
```

Для быстрой вставки инструкции `function` в редакторе VS Code предназначена следующая комбинация:

```
function =>  
function name(params) {  
  
}
```

10.2. Расположение определений функций

В языке JavaScript определение функции может быть расположено после ее вызова:

```
test(); // Это функция test()  
function test() {  
    console.log('Это функция test()');  
}
```

Но традиционно определения функций, как правило, записываются перед первыми их вызовами, как показано в листинге 10.1.

Листинг 10.1. Расположение функций

```
function showOK() {  
    console.log('Сообщение при удачно выполненной операции');  
}
```



```
function showMessage(msg) {  
    console.log(msg);  
}  
function sum(x, y) {  
    return x + y;  
}  
showOK(); // Сообщение при удачно выполненной операции  
showMessage('Сообщение'); // Сообщение  
let n = sum(5, 2); // Переменной n будет присвоено значение 7  
showMessage(`n = ${n}`); // n = 7
```

10.3. Вложенные функции

Определение одной функции можно вложить внутрь тела другой функции. В этом случае вложенная функция будет видна только внутри внешней функции. Причем вложенная функция имеет доступ к локальным переменным и параметрам внешней функции (листинг 10.2).

Листинг 10.2. Вложенные функции

```
function test(x) { // Внешняя функция  
    let y = 10;  
    function func() { // Вложенная функция  
        console.log(`x = ${x}; y = ${y}`);  
    }  
    func(); // Вызов вложенной функции  
}  
test(5); // Вызов внешней функции  
// x = 5; y = 10
```

Как видно из результата, внутри вложенной функции мы имеем доступ к параметру внешней функции, а также к объявленной в ней локальной переменной.

10.4. Замыкание

Ссылку на вложенную функцию можно вернуть в качестве значения внешней функции. В этом случае имя вложенной функции указывается в операторе `return` без круглых скобок. Вложенные функции захватывают переменные в родительской области видимости. После возврата ссылки на вложенную функцию и выхода из внешней функции область видимости сохраняется, и мы можем получить доступ к значениям переменных, существовавших в момент создания вложенной функции (листинг 10.3). Такая возвращенная в качестве параметра функция называется *замыканием*.

Листинг 10.3. Замыкание

```
function test(x) {    // Внешняя функция
  function func(y) { // Вложенная функция
    console.log(`x = ${x}; y = ${y}`);
  }
  return func;        // Возврат ссылки на вложенную функцию
}

let f1 = test(5);      // Вызов внешней функции
f1(10);                // Вызов вложенной функции
// x = 5; y = 10
f1(55);                // Вызов вложенной функции
// x = 5; y = 55
let f2 = test(2);      // Вызов внешней функции
f2(14);                // Вызов вложенной функции
// x = 2; y = 14
f2(88);                // Вызов вложенной функции
// x = 2; y = 88
test(3)(6);            // Вызов внешней и вложенной функций
// x = 3; y = 6
```

10.5. Класс *Function*

Класс `Function` позволяет создавать функцию как объект этого класса. Делается это таким образом:

```
<Переменная> = new Function([<Параметр 1>[, ... , <Параметр N>],]
                           <Тело функции>)
```

Например, функцию суммирования двух чисел

```
function sum(x, y) {
  return x + y;
}
```

можно переписать так (оператор `new` можно не указывать):

```
let sum = new Function('x', 'y', 'return x + y');
console.log( typeof sum ); // function
console.log( sum(5, 7) ); // 12
```

Указывать тело функции в виде строки очень неудобно. По этой причине данным способом никто не пользуется.

Каждая функция содержит следующие методы из класса `Function`:

- `call([<Объект>[, <Значения через запятую>]])` — вызывает функцию в контексте объекта `<Объект>` и передает ей значения, указанные во втором и последующих параметрах. Объект, указанный в первом параметре метода `call()`, доступен внутри функции через указатель `this`. Пример:

```
function sum(x, y) {
  console.log(this); // { a: 1 }
  return x + y;
}
let obj = { a: 1 };
console.log( sum.call(obj, 5, 7) ); // 12
```

Если первый параметр имеет значение `undefined` или `null`, то в обычном режиме функция вызывается в глобальном контексте:

```
function sum(x, y) {
  console.log(this); // Object [global] {...}
  return x + y;
}
console.log( sum.call(null, 5, 7) ); // 12
```

В строгом режиме в этом случае указатель `this` будет иметь значение `undefined` или `null`:

```
'use strict';
function sum(x, y) {
  console.log(this); // undefined или null
  return x + y;
}
console.log( sum.call(undefined, 5, 7) ); // 12
console.log( sum.call(null, 5, 5) ); // 10
```

- `apply([<Объект>[, <Массив с параметрами>]])` — метод аналогичен методу `call()`, но значения параметров вызываемой функции указываются в виде массива:

```
function sum(x, y) {
  console.log(this);
  return x + y;
}
let obj = { a: 1 };
console.log( sum.apply(obj, [5, 7]) ); // 12
// { a: 1 }
console.log( sum.apply(null, [5, 7]) ); // 12
// В обычном режиме: Object [global]
// В строгом режиме: null
```

Для вызова функции с передачей контекста также можно воспользоваться статическим методом `apply(<Функция>, <Объект>, <Массив с параметрами>)` из класса `Reflect` (см. разд. 11.11):

```
function sum(x, y) {
  console.log(this);
  return x + y;
}
console.log( Reflect.apply(sum, undefined, [30, 10]) ); // 40
// В обычном режиме: Object [global]
// В строгом режиме: undefined
```

```
console.log( Reflect.apply(sum, { a: 10 }, [ 15, 6 ]) ); // 21
// В обычном режиме: { a: 10 }
// В строгом режиме: undefined
```

- ❑ `bind(<Объект>[, <Параметры через запятую>])` — создает новый объект функции в контексте указанного в первом параметре объекта, который доступен внутри функции через указатель `this`:

```
global.a = 22;
let obj = {
  a: 33,
  getA: function() {
    console.log(this);
    if (this === undefined) return undefined;
    return this.a;
  }
};
// this - глобальный контекст или undefined в строгом режиме
let func = obj.getA;
console.log( func() ); // 22 (или undefined в строгом режиме)
// this - контекст obj
let funcGetA = func.bind(obj);
console.log( funcGetA() ); // 33
```

10.6. Функции с произвольным числом параметров

Получить произвольное число параметров в функции можно посредством объекта `arguments` и оператора `REST`.

10.6.1. Объект *arguments*

Внутри функции доступен объект `arguments`, который позволяет получить доступ ко всем параметрам, переданным функции. Этот объект поддерживает функциональность массива, следовательно, получить доступ к параметру можно, указав его индекс внутри квадратных скобок. Свойство `length` хранит число параметров, переданных функции. Пример:

```
function sum(x, y) {
  console.log(arguments.length); // 2
  return arguments[0] + arguments[1];
}
console.log( sum(5, 6) ); // 11
```

При использовании объекта `arguments` можно передать функции больше параметров, чем было первоначально объявлено. Например, можно просуммировать сразу несколько чисел, а не только два (листинг 10.4).

Листинг 10.4. Произвольное число параметров

```
function sum(x, y) {
  let z = 0;
  for (let i = 0; i < arguments.length; i++) {
    z += arguments[i];
  }
  return z;
}
console.log( sum(5, 6, 7, 20) ); // 38
```

10.6.2. Оператор *REST*

Избыточные параметры, переданные функции при вызове, можно поместить в массив (листинг 10.5). Для этого перед именем последнего параметра указываются три точки (оператор *REST*).

Листинг 10.5. Оператор *REST*

```
function sum(x, y, ...arr) {
  let z = x + y;
  for (let i = 0; i < arr.length; i++) {
    z += arr[i];
  }
  return z;
}
console.log( sum(5, 6, 7, 20) ); // 38
```

Первые два значения будут присвоены переменным *x* и *y*, а все остальные будут добавлены в массив *arr*.

10.7. Способы передачи параметров в функцию

При вызове в функцию передается копия значения переменной. Таким образом, изменение значения внутри функции не затронет значение исходной переменной.

Пример:

```
function test(x) {
  x = 10;
}
let x = 20;
test(x);
console.log('x = ' + x); // x = 20
```

Однако при передаче функции объекта фактически передается ссылка на этот объект, а не сам объект. Внутри функции мы не можем присвоить переменной ссылку на другой объект, а вот изменить свойства переданного объекта можем. Пример:

```
function test(arr) {  
    arr[0] = 33;  
}  
let a = [1, 2];  
test(a);  
console.log(a); // [ 33, 2 ]
```

Если функция принимает отдельные параметры или указан оператор `REST`, а данные хранятся в массиве, то можно распаковать массив (листинг 10.6), указав перед ним три точки (оператор `SPREAD`).

Листинг 10.6. Оператор `SPREAD`

```
function sum(x, y, ...arr) {  
    let z = x + y;  
    for (let i = 0; i < arr.length; i++) {  
        z += arr[i];  
    }  
    return z;  
}  
console.log( sum( 5, 6, 7, 20 ) ); // 38  
console.log( sum( ...[5, 6, 7, 20] ) ); // 38  
console.log( sum( 5, 6, ...[7, 20] ) ); // 38
```

10.8. Необязательные параметры

Некоторые параметры можно сделать необязательными. Для этого используется проверка равенства необязательного параметра значению `undefined` или оператор `||` (если первое выражение не может быть преобразовано в `true`, то возвращается значение второго выражения). Пример:

```
function test(str) {  
    str = str || 'Значение по умолчанию';  
    return str;  
}  
console.log( test() ); // Значение по умолчанию  
console.log( test('Значение указано') ); // Значение указано
```

Еще значение по умолчанию можно указать при объявлении параметра, после его имени через знак `=`:

```
function test(str='Значение по умолчанию') {  
    return str;  
}
```

```
console.log( test() ); // Значение по умолчанию
console.log( test('Значение указано') ); // Значение указано
```

Следует учитывать, что значение параметра по умолчанию вычисляется при каждом вызове функции, поэтому в следующем примере каждый раз будет создаваться новый массив:

```
function test(x, arr=[]) {
    arr.push(x);
    return arr;
}
console.log( test(2) ); // [ 2 ]
console.log( test(5) ); // [ 5 ]
```

10.9. Анонимные функции

Функция может вообще не иметь имени. В этом случае ссылку на анонимную функцию сохраняют в переменной. Пример:

```
const func = function() { // Присваиваем ссылку на анонимную функцию
    console.log('Сообщение');
};
func(); // Вызываем анонимную функцию через переменную func
```

Анонимная функция становится видимой только после определения. Если вызов функции поместить перед определением, будет выведено сообщение об ошибке. При использовании обычных функций ошибки не будет.

Ссылку на вложенную функцию можно вернуть в качестве значения в конструкции `return`. Чтобы вызвать вложенную функцию, круглые скобки указываются два раза.

Пример:

```
const func = function() { // Присваиваем ссылку на анонимную функцию
    return function() {    // Возвращаем ссылку на вложенную функцию
        console.log('Это вложенная функция');
    };
};
func()(); // Вызываем вложенную функцию через переменную func
```

При использовании анонимных функций следует учитывать, что при указании внутри функции глобальной переменной будет сохранена ссылка на эту переменную, а не на ее значение:

```
let x = 5;
const func = function() {
    return x; // Сохраняется ссылка, а не значение переменной x !
};
console.log( func() ); // 5
x = 10; // Изменили значение
console.log( func() ); // 10, а не 5
```

Анонимные функции захватывают переменные в родительской области видимости (это называется замыканием). Если анонимная функция объявлена внутри функции, то после выхода из функции область видимости сохраняется и мы можем получить доступ к переменным, существовавшим в момент создания анонимной функции.

Пример:

```
function test() {
  let x = 10, y = 5;
  return function() {
    return x + y;
  };
}

let func = test();
console.log( func() ); // 15
```

Любопытно, что анонимной функции можно дать имя, но оно будет видимо только внутри анонимной функции. Имя можно использовать для рекурсивного вызова функции, а также в целях отладки. Пример указания имени:

```
const func = function myFunction() {
  console.log('Сообщение');
  console.log(myFunction); // [Function: myFunction]
};

func();
// myFunction();
// ReferenceError: myFunction is not defined
```

Анонимная функция удобна, когда какой-либо функции требуется в качестве параметра передать ссылку на другую функцию:

```
function func(x, y, fn) {
  console.log( fn(x, y) );
}

const funcAdd = function(x, y) {
  return x + y;
};

const funcSub = function(x, y) {
  return x - y;
};

func(10, 5, funcAdd); // 15
func(10, 5, funcSub); // 5
func(10, 5, function(x, y) {
  return x / y;
}); // 2
```


10.10. Стрелочные функции (лямбда-выражения)

В языке JavaScript поддерживаются *стрелочные функции* (в других языках они называются *лямбда-выражениями*). Объявление стрелочной функции имеет следующие форматы:

```
<Переменная> = <Параметр> => <Возвращаемое значение>;  
<Переменная> = ([<Параметр 1>[, . . . , <Параметр N>]]) =>  
    <Возвращаемое значение>;  
<Переменная> = ([<Параметр 1>[, . . . , <Параметр N>]]) => {  
    <Тело функции>  
    [return <Возвращаемое значение>;]  
};
```

Стрелочные функции, как и анонимные, не имеют имени. Ссылка на функцию сохраняется в переменной. Чтобы вызвать функцию, нужно после имени переменной указать круглые скобки и внутри них перечислить значения параметров. Пример:

```
const func = (x, y) => { return x + y };  
console.log( func(5, 7) ); // 12
```

Если тело стрелочной функции состоит из одного выражения, фигурные скобки можно не указывать. Результат вычисления этого выражения станет возвращаемым значением. Пример:

```
const func = (x, y) => x + y;  
console.log( func(5, 7) ); // 12
```

Если стрелочная функция принимает только один параметр, круглые скобки можно не указывать:

```
const func = n => n * 2;  
console.log( func(5) ); // 10
```

Круглые скобки обязательно указываются при отсутствии параметров:

```
const func = () => console.log('Сообщение');  
func(); // Сообщение
```

Если возвращаемым значением является объект, записанный в виде литерала, его следует заключить в круглые скобки:

```
const func = () => ({ a: 1 });  
console.log( func() ); // { a: 1 }
```

Стрелочные функции захватывают переменные в родительской области видимости. Если стрелочная функция объявлена внутри функции, то после выхода из функции область видимости сохраняется и мы можем получить доступ к переменным, существовавшим в момент создания стрелочной функции. Пример:

```
function test() {  
    let x = 10, y = 5;
```

```
    return () => x + y;
  }
  let func = test();
  console.log( func() ); // 15
```

В отличие от обычных функций, внутри стрелочных функций нет доступа к объекту `arguments`. Чтобы передать произвольное число параметров, нужно вставить оператор `REST` перед именем параметра. Пример:

```
const sum = (...arr) => {
  let z = 0;
  for (let i = 0; i < arr.length; i++) {
    z += arr[i];
  }
  return z;
};
console.log( sum(5, 6, 7, 20) ); // 38
```

В отличие от анонимных функций, стрелочные функции сохраняют ссылку на родительский указатель `this`:

```
function test() {
  console.log(this); // { a: 1 }
  let self = this;
  let func1 = function() {
    console.log(this);
    // В обычном режиме: Object [global] {...}
    // В строгом режиме: undefined
    console.log(self); // { a: 1 }
  };
  let func2 = () => console.log(this); // { a: 1 }
  func1();
  func2();
}
let obj = { a: 1 };
test.call(obj);
```

Причем указание контекста в методах `call()` и `apply()` не изменяет значение указателя `this`, т. к. стрелочные функции не содержат собственного указателя `this`, а захватывают его из родительской области видимости.

10.11. Функции-генераторы

В языке JavaScript поддерживаются *функции-генераторы*. Функцией-генератором называется функция, которая при очередном вызове возвращает очередное значение из заданного набора. Причем после возврата значения выполнение такой функции не завершается, а приостанавливается, а при следующем вызове — продолжается с того места, на котором было приостановлено.

При определении функции-генератора после ключевого слова `function` указывается символ звездочка (`function*`). Для возврата очередного значения и приостановки

выполнения функции применяется оператор `yield` <Возвращаемое значение>, а не `return`.

В качестве примера напишем функцию-генератор, которая при каждом вызове возвращает очередное число из диапазона от 1 до указанного в первом параметре, умноженное на значение из второго параметра (листинг 10.7).

Листинг 10.7. Функции-генераторы

```
function* test(x, y) {
  for (let i = 1; i <= x; i++) {
    yield i * y;
  }
}

let obj = test(10, 2);
for (let n of obj) {
  console.log(n);
}

obj = test(3, 2);
console.log(obj.next().value); // 2
console.log(obj.next().value); // 4
console.log(obj.next().value); // 6
console.log(obj.next().value); // undefined
obj = test(3, 2);
console.log( [...obj] );      // [ 2, 4, 6 ]
```

Генератор позволяет сгенерировать бесконечное количество значений (по одному при каждом вызове).

Функция-генератор при каждом вызове фактически возвращает объект. Он содержит метод `next()`, возвращающий объект со свойствами `value` (хранит текущее значение) и `done` (хранит значение `true`, если больше нет значений, и `false` — в противном случае). Пример:

```
obj = test(3, 2);
console.log( obj.next() ); // { value: 2, done: false }
console.log( obj.next() ); // { value: 4, done: false }
console.log( obj.next() ); // { value: 6, done: false }
console.log( obj.next() ); // { value: undefined, done: true }
```

Метод `next()` также позволяет передать параметр, который вернет оператор `yield` при следующем вызове генератора:

```
function* gen() {
  console.log(yield);
  console.log(yield);
}

let obj = gen();
obj.next();
obj.next('1');      // 1
obj.next('2');      // 2
```

Оператор `yield` позволяет делегировать дальнейшее выполнение другому генератору. В этом случае после оператора `yield` указывается символ `*`. Пример:

```
function* gen1() {
  yield 0;
  yield* gen2();
}
function* gen2() {
  yield 1;
  yield 2;
}
function* gen3() {
  yield* [1, 2, 3];
}
let obj = gen1();
console.log(obj.next().value); // 0
console.log(obj.next().value); // 1
console.log(obj.next().value); // 2
obj = gen3();
console.log(obj.next().value); // 1
console.log(obj.next().value); // 2
console.log(obj.next().value); // 3
```

10.12. Рекурсия

Рекурсия — это вызов функцией самой себя. При каждом вызове функции создается новый набор локальных переменных. Рекурсию удобно использовать для перебора объекта, имеющего заранее неизвестную структуру, или выполнения неопределенного количества операций. Типичное применение рекурсии — вычисление факториала числа (листинг 10.8).

Листинг 10.8. Вычисление факториала

```
function factorial(x) {
  if (x <= 1) return 1;
  return (x * factorial(x - 1));
}
console.log( factorial(3) ); // 6
console.log( factorial(4) ); // 24
console.log( factorial(5) ); // 120
console.log( factorial(6) ); // 720
console.log( factorial(7) ); // 5040
```

Пример использования имени, указанного в анонимной функции:

```
const factorial = function fact(x) {
  if (x <= 1) return 1;
  return (x * fact(x - 1));
}
console.log( factorial(7) ); // 5040
```



ГЛАВА 11

Классы и объекты

Объектно-ориентированное программирование (ООП) — это способ организации программы, при котором предметы реального мира описываются в виде объектов, создаваемых на основе различных классов.

Основным "кирпичиком" ООП является *класс*. Класс включает набор переменных (*свойств*) и функций для управления этими переменными (*методов*). В совокупности свойства и методы называются *членами* класса. На основе класса можно создать произвольное количество *объектов* (иначе называемых *экземплярами класса*). Каждый объект хранит в своих свойствах собственный набор данных и является независимым от остальных объектов.

11.1. Объекты

В языке JavaScript объект можно создать без описания класса с помощью фигурных скобок или класса `Object`. Оба способа эквиваленты.

11.1.1. Создание объекта с помощью фигурных скобок

Синтаксис создания объекта с помощью фигурных скобок таков:

```
<Переменная> = {[
  <Свойство 1>: <Значение 1>[,
  ...
  <Свойство N>: <Значение N>]
}];
```

Если между фигурными скобками нет никаких инструкций, создается пустой объект:

```
let obj = {}; // Пустой объект
console.log(obj); // {}
console.log(typeof obj); // object
```

При создании объектов следует учитывать один очень важный момент. Например, нам необходимо определить два пустых объекта, которые в дальнейшем будут использоваться отдельно. Очень силен соблазн написать код следующим образом:

```
let obj1, obj2;  
obj1 = obj2 = {};           // Якобы определили два объекта
```

Проблема заключается в том, что будет создан только один объект, а в переменных будут сохранены лишь ссылки на него. Таким образом, все изменения `obj1` станут отражаться и на переменной `obj2`:

```
let obj1, obj2;  
obj1 = obj2 = {};           // Якобы определили два объекта  
obj1.test = 'Это значение свойства test объекта obj1';  
console.log(obj2.test);  
// Выведет: Это значение свойства test объекта obj1
```

Помните, что при присваивании объекта копируется ссылка на него, а не сам этот объект. Поэтому создавать объекты, даже полностью одинаковые, необходимо раздельно. Пример:

```
let obj1 = {}, obj2 = {};  
obj1.test = 'Это значение свойства test объекта obj1';  
console.log(obj2.test); // Выведет: undefined
```

Переменная может содержать специальное значение `null`, которое означает нулевую ссылку, не указывающую ни на один объект. Это значение удобно возвращать из функции при невозможности создать объект, например в случае ошибки. Пример:

```
let obj = null;              // Нет объекта  
console.log(obj);            // null  
console.log(typeof obj);     // object
```

11.1.2. Свойства объекта

Для добавления свойства в объект следует внутри фигурных скобок указать его имя и после двоеточия задать значение. Если нужно добавить несколько свойств, то пары `свойство: значение` перечисляются через запятую. Пример:

```
let car = {  
  model: 'BA3-2109',  
  year: 2007  
};  
console.log(car); // { model: 'BA3-2109', year: 2007 }
```

В конце описания последнего свойства можно поставить запятую. Это может оказаться полезным, если в объект планируется в дальнейшем добавлять другие свойства. Пример указания завершающей запятой:

```
let car = {  
  model: 'BA3-2109',  
  year: 2007,  
};  
console.log(car); // { model: 'BA3-2109', year: 2007 }
```

После создания объекта ссылка на него сохраняется в переменной. Используя ссылку, можно обращаться к свойствам, получая или изменяя их значения. Форматы обращения к свойствам:

```
<Переменная> = <Объект>.<Имя свойства>  
<Объект>.<Имя свойства> = <Значение>
```

Получим значение свойства `model` и изменим значение свойства `year`:

```
// Получаем значение свойства  
let model = car.model;  
console.log(model);      // ВАЗ-2109  
// Изменяем значение свойства  
car.year = 2009;  
console.log(car);        // { model: 'ВАЗ-2109', year: 2009 }
```

ОБРАТИТЕ ВНИМАНИЕ

Хотя имена переменной и свойства объекта совпадают, никакого конфликта имен не будет, так как переменная и свойство расположены в разных областях видимости.

Фигурные скобки ограничивают область видимости, а для доступа к именам в этой области видимости нужно использовать точечную нотацию, предварительно указав имя объекта.

Имя свойства должно быть допустимым идентификатором. Если имя содержит запрещенные символы (например, дефис или пробел), его следует указать в виде строки. Пример:

```
let car = {  
  'Модель автомобиля': 'ВАЗ-2109',  
  'Год выпуска': 2007  
};  
console.log(car);  
// { 'Модель автомобиля': 'ВАЗ-2109', 'Год выпуска': 2007 }
```

Кроме того, допускается динамическое создание имени свойства на основе значения какой-либо переменной или даже результата, возвращаемого выражением. В этом случае переменная или выражение указывается внутри квадратных скобок. Пример:

```
let name = 'test';  
let obj = {  
  [name]: 20,  
  ['prop' + 2]: 30  
}  
console.log(obj); // { test: 20, prop2: 30 }
```

Для обращения к свойствам в последних двух случаях удобно использовать следующие форматы:

```
<Переменная> = <Объект>[<Имя свойства>]  
<Объект>[<Имя свойства>] = <Значение>
```

Внутри квадратных скобок можно указать имя свойства в виде строки, переменной или выражения:

```
console.log( car['Модель автомобиля'] ); // ВАЗ-2109
car['Год выпуска'] = 2009;
console.log(car);
// { 'Модель автомобиля': 'ВАЗ-2109', 'Год выпуска': 2009 }

console.log( obj[name] ); // 20
obj['prop' + 2] = 40
console.log(obj);           // { test: 20, prop2: 40 }
```

Если нужно добавить свойство, имя которого совпадает с именем уже существующей переменной, и занести в свойство значение из этой переменной, то вместо пары `свойство: значение` указывается только имя переменной:

```
let name = 'Николай';
let obj = {
  name,
  age: 30
}
console.log(obj); // { name: 'Николай', age: 30 }
```

При присваивании значения несуществующему свойству оно будет создано:

```
let car = {};
car.model = 'ВАЗ-2109';
car.year = 2007;
car.color = 'red';
console.log(car); // { model: 'ВАЗ-2109', year: 2007, color: 'red' }
```

При объявлении переменной объектного типа с помощью ключевого слова `const` следует помнить, что присвоить новое значение переменной будет нельзя, но можно изменить значения свойств объекта:

```
const obj = { a: 1, b: 2 };
console.log(obj); // { a: 1, b: 2 }
// Присвоить другое значение нельзя
// obj = {}; // Ошибка
// TypeError: Assignment to constant variable.
// Значение свойства можно изменить!
obj.a = 88;
console.log(obj); // { a: 88, b: 2 }
```

Свойство объекта в качестве значения может хранить другой объект (который носит название *вложенного*). Для доступа к свойствам вложенного объекта следует через точки записать имя переменной с объектом, имя свойства объекта, хранящего вложенный объект, и имя нужного свойства вложенного объекта. Пример:

```
// Свойство с объекта obj хранит вложенный объект
const obj = { a: 1, b: 2, c: { d: 3, e: 4 } };
```



```
// Выводим вложенный объект
console.log(obj.c);           // { d: 3, e: 4 }
// Выводим значение свойства d вложенного объекта
console.log(obj.c.d);         // 3
// Заносим новое значение в свойство d вложенного объекта
obj.c.d = 8;
console.log(obj.c.d);         // 8
```

При попытке обратиться к свойству вложенного объекта, хранящегося в несуществующем свойстве внешнего объекта, будет сгенерировано исключение `TypeError`:

```
const obj = { a: 1, b: 2, c: { d: 3, e: 4 } };
// Пытаемся получить значение свойства d вложенного объекта из
// несуществующего свойства t внешнего объекта
// console.log(obj.t.d);
// TypeError: Cannot read property 'd' of undefined
```

В таком случае пригодится оператор опциональной последовательности `?`, указываемый сразу после имени свойства внешнего объекта. Если свойство с таким именем существует и хранит вложенный объект, оператор возвращает значение свойства вложенного объекта, имя которого указано далее через точку. Если же такого свойства нет или оно хранит не объект, оператор возвращает `undefined` без генерирования какого-либо исключения. Пример:

```
const obj = { a: 1, b: 2, c: { d: 3, e: 4 } };
// Выводим значение свойства d вложенного объекта из существующего
// свойства c внешнего объекта — успешно!
console.log(obj.c?.d);         // 3
// При попытке получить значение свойства d вложенного объекта
// из несуществующего свойства t внешнего объекта получим значение
// undefined без генерирования исключения
console.log(obj.t?.d);         // undefined
```

11.1.3. Распаковка объекта

Для сохранения значений свойств в переменных можно использовать следующий формат:

```
{<Свойство 1>[: <Имя>][ = <Значение>][, ..., <Свойство N>]} = <Объект>
```

Пример:

```
let car = {
  model: 'BA3-2109',
  year: 2007,
  color: 'red'
};
let model1 = car.model;
console.log(model1);      // BA3-2109
let {model, year, color} = car;
```

```
console.log(model);    // ВАЗ-2109
console.log(year);     // 2007
console.log(color);    // red
```

Необязательно указывать все имена свойств — достаточно перечислить только нужные. По умолчанию имя свойства становится именем переменной. Если необходимо изменить имя создаваемой переменной, это имя следует записать через двоеточие после имени свойства. Пример:

```
let {model: m, year: y} = car;
console.log(`${m}; ${y}`); // ВАЗ-2109; 2007
```

Кроме того, после оператора = можно указать значение по умолчанию, если свойства не существует или оно имеет значение `undefined`:

```
let obj = { a: 1, b: 2, c: undefined };
let {a, b: b2, c: c2 = 10, d = 20} = obj;
console.log(`${a}; ${b2}; ${c2}; ${d}`); // 1; 2; 10; 20
```

Если перед последней переменной указать три точки, то в этой переменной будет сохранен объект с оставшимися свойствами, не перечисленными в списке:

```
let obj = { a: 1, b: 2, c: 3 };
let {a, ...obj2} = obj;
console.log(a);    // 1
console.log(obj2); // { b: 2, c: 3 }
```

Чтобы получить значение свойства с программно формируемым именем, это имя следует заключить в квадратные скобки. После двоеточия нужно указать имя создаваемой переменной. Пример:

```
let obj = { a: 1, b: 2, c: 3 };
let key = 'a';
let { [key]: a2 } = obj;
console.log(a2 );    // 1
console.log( obj[key] ); // 1
```

Пример распаковки вложенного объекта:

```
let obj = { a: { b: 2, c: undefined } };
let {a: obj2} = obj;
console.log(obj2); // { b: 2, c: undefined }
let {a: { b = 1, c: n = 88 } } = obj;
console.log(b);    // 2
console.log(n);    // 88
```

Операция распаковки допустима и в параметрах функции:

```
function test( {a, c: n = 88} ) {
    console.log(a);    // 1
    console.log(n);    // 88
}
let obj = { a: 1, b: 2, c: undefined };
test(obj);
```

11.1.4. Методы объекта

В качестве значения свойству можно присвоить ссылку на функцию, после чего свойство станет *методом объекта*:

```
<Свойство>: function [<Имя>]([<Параметры>]) { ... }
```

Обращение к методам осуществляется аналогично обращению к свойствам, только после имени метода необходимо указать круглые скобки:

```
<Объект>.<Имя метода>([<Параметры>])
```

Для обращения к методам также можно использовать квадратные скобки, внутри которых указывается имя в виде строки, переменная или выражение:

```
<Объект>[<Имя метода>](<Параметры>)
```

Пример:

```
let car = {
  model: 'BA3-2101',
  getModel: function() {
    return this.model;
  },
  setModel: function setModel(name) {
    this.model = name;
  }
};
car.setModel('BA3-2109');
console.log( car.model );           // BA3-2109
console.log( car.getModel() );      // BA3-2109
car['setModel']('LADA Granta');
console.log( car['getModel']() );   // LADA Granta
```

Существует альтернативный способ определения методов, при котором ключевое слово `function` и двоеточие не указываются:

```
let car = {
  model: 'BA3-2101',
  getModel() {
    return this.model;
  },
  setModel(name) {
    this.model = name;
  }
};
car.setModel('BA3-2109');
console.log( car.getModel() );      // BA3-2109
car['setModel']('LADA Granta');
console.log( car['getModel']() );   // LADA Granta
```

Определение методов-генераторов осуществляется следующими способами:

```
let obj = {
  arr: [1, 2, 3],
```

```
gen1: function* () {
  for (let i = 0; i < this.arr.length; i++) {
    yield this.arr[i];
  }
},
*gen2() {
  for (let i = 0; i < this.arr.length; i++) {
    yield this.arr[i];
  }
}
};
let iter = obj.gen1();
console.log(iter.next().value); // 1
console.log(iter.next().value); // 2
console.log(iter.next().value); // 3
console.log(iter.next().value); // undefined
iter = obj.gen2();
for (const value of iter) {
  console.log(value);
}
```

При попытке вызвать несуществующий метод у объекта будет сгенерировано исключение `TypeError`:

```
const obj = { a: 1, b: 2 };
// Пытаемся вызвать несуществующий метод z()
console.log(obj.z());
// TypeError: obj.z is not a function
```

Избежать этого можно, воспользовавшись оператором опциональной последовательности (см. *разд. 11.1.2*), который в случае методов записывается так: `?.` и ставится между именем метода и открывающей круглой скобкой. Пример:

```
const obj = { a: 1, b: 2 };
// Пытаемся вызвать несуществующий метод z() и получаем undefined без
// генерирования исключения
console.log(obj.z?.()); // undefined
```

Также можно вызывать методы из вложенных объектов (о вложенных объектах рассказывалось в *разд. 11.1.2*). Пример вызова таких методов приведен в листинге 11.1.

Листинг 11.1. Вызов методов вложенного объекта

```
const obj = {
  a: 1,
  b: 2,
  c: {
    d: 3,
    e: 4,
```

```

        method() {
            return this.d * this.e;
        }
    }
};
// Вызываем метод method() вложенного объекта из свойства с внешнего
// объекта
console.log( obj.c.method() ); // 12
// Пытаемся вызвать метод method() вложенного объекта из несуществующего
// свойства z внешнего объекта — получаем значение undefined без
// генерирования исключения
console.log( obj.z?.method() ); // undefined

```

11.1.5. Указатель *this*

Внутри метода доступен указатель *this*, ссылающийся на текущий объект. С помощью этого указателя можно получить доступ к свойствам и методам текущего объекта. Пример:

```

<Переменная> = this.<Имя свойства>
this.<Имя свойства> = <Значение>
<Переменная> = this['<Имя свойства>']
this['<Имя свойства>'] = <Значение>

```

Пример обращения к свойству *model* внутри метода *getModel()*:

```

let car = {
    model: 'BA3-2109',
    getModel() {
        return this.model; // return this['model'];
    }
};

```

Если в данном примере указатель *this* удалить, то получим ошибку обращения к неопределенной локальной переменной:

```
ReferenceError: model is not defined
```

Для определения методов нельзя использовать стрелочные функции (лямбда-выражения), т. к. они не содержат собственного указателя *this*, а захватывают его из родительской области видимости. Поэтому они не могут иметь доступа к свойствам объекта. Пример:

```

let car = {
    model: 'BA3-2109',
    getModel: () => {
        console.log(this); // {}
        return this.model;
    }
};
console.log( car.getModel() ); // undefined

```

На самом деле при использовании функций указатель `this` потерять очень просто. Достаточно присвоить ссылку на метод какой-либо переменной, а потом вызвать метод через переменную. Пример:

```
let car = {
  model: 'BA3-2109',
  getModel() {
    return this.model;
  }
};

console.log( car.getModel() ); // BA3-2109
let func = car.getModel;
// Потеряли this
console.log( func() );          // undefined в обычном режиме
```

В строгом режиме указатель `this` будет содержать значение `undefined`, поэтому получим исключение:

```
TypeError: Cannot read property 'model' of undefined
```

В этом случае следует передать контекст явным образом с помощью методов `call()`, `apply()` или `bind()` (см. *разд. 10.5*):

```
console.log( func.call(car) ); // BA3-2109
console.log( func.apply(car) ); // BA3-2109
let boundFunc = func.bind(car);
console.log( boundFunc() );     // BA3-2109
```

Аналогичная ситуация возникает при передаче ссылки на метод в функции таймеров:

```
let car = {
  model: 'BA3-2109',
  print() {
    console.log(this.model);
  }
};

setTimeout(car.print, 2000);          // undefined
setTimeout(car.print.bind(car), 2000); // BA3-2109
```

Потерять указатель `this` можно также внутри вложенной в метод функции:

```
let car = {
  model: 'BA3-2109',
  func() {
    let print = function() {
      console.log(this.model);
    };
    print();
  }
};
```

```
car.func();           // undefined в обычном режиме
car.func.call(car);   // undefined в обычном режиме
car.func.apply(car);  // undefined в обычном режиме
```

В строгом режиме указатель `this` будет содержать значение `undefined`, поэтому получим исключение:

```
TypeError: Cannot read property 'model' of undefined
```

В этом случае методы `call()`, `apply()` и `bind()` нам не помогут. Существует несколько способов решения проблемы. Во-первых, можно передать указатель `this` в качестве параметра при вызове вложенной функции. Пример:

```
let car = {
  model: 'BA3-2109',
  func() {
    let print = function(self) {
      console.log(self.model);
    };
    print(this);
  }
};
car.func();           // BA3-2109
```

Во-вторых, можно воспользоваться замыканием, присвоив указатель `this` локальной переменной внутри метода:

```
let car = {
  model: 'BA3-2109',
  func() {
    let self = this;
    let print = function() {
      console.log(self.model);
    };
    print();
  }
};
car.func();           // BA3-2109
```

11.1.6. Setter и getter

Ключевое слово `set` связывает свойство с функцией (setter), которая будет вызвана при попытке присвоить свойству новое значение. Формат:

```
set <Имя свойства>(<Значение>) {
  // ...
}
```

Функция принимает только один параметр, в который передается новое значение, присваиваемое свойству:

```
<Объект>.<Имя свойства> = <Значение>;
```

Ключевое слово `get` связывает свойство с функцией (getter), которая будет вызвана при попытке получения значения этого свойства:

```
get <Имя свойства>() {  
    return ...;  
}
```

Функция не должна принимать никаких параметров и возвращать результат, который и станет значением этого свойства:

```
<Переменная> = <Объект>.<Имя свойства>;
```

Пример использования ключевых слов `set` и `get`:

```
let car = {  
    set model(name) {  
        this._model = name;  
    },  
    get model() {  
        return this._model;  
    }  
};  
console.log(car);           // { model: [Getter/Setter] }  
car.model = 'BA3-2109'; // Вызывается set model(name)  
console.log(car.model); // Вызывается get model()  
// BA3-2109  
// Можно получить доступ к _model напрямую  
console.log(car._model); // BA3-2109
```

Допускается динамическое создание имени свойства. В этом случае имя свойства указывается внутри квадратных скобок. Пример:

```
let nameProp = 'model';  
let car = {  
    set [nameProp](name) {  
        this._model = name;  
    },  
    get [nameProp]() {  
        return this._model;  
    }  
};  
car.model = 'BA3-2109';  
console.log( car.model ); // BA3-2109  
car[nameProp] = 'BA3-2109';  
console.log( car[nameProp] ); // BA3-2109
```

Чтобы добавить `getter` и `setter` после создания объекта, следует воспользоваться статическим методом `defineProperty(<Объект>, <Имя свойства>, <Настройки>)` из класса `Object`:

```
let car = {  
    _model: ''  
};
```



```
Object.defineProperty(car, 'model', {
  set: function(name) {
    this._model = name;
  },
  get: function() {
    return this._model;
  }
});
car.model = 'BA3-2109';
console.log( car.model );      // BA3-2109
console.log( car._model );     // BA3-2109
```

11.1.7. Свойство `__proto__`

Каждый объект содержит специальное свойство `__proto__`, которое позволяет получить доступ для чтения и записи к прототипу объекта (обозначается в спецификации как `[[Prototype]]`). С помощью этого свойства мы можем расширить объект за счет наследования свойств и методов другого объекта. Пример:

```
let objA = { a: 10 };
let objB = { b: 55 };
objB.__proto__ = objA;
console.log(objB.a);          // 10
console.log(objB.b);          // 55
```

В этом примере объект `objB` наследует свойства объекта `objA`. С помощью точечной нотации мы можем получить доступ как к собственному свойству `b`, так и к унаследованному свойству `a`.

Свойство `__proto__` также можно указать внутри фигурных скобок. Создадим объект `objC`, который наследует все свойства объектов `objB` и `objA`:

```
let objC = {
  b: 22,
  c: 32,
  __proto__: objB
};
console.log(objC.a);          // 10
console.log(objC.b);          // 22, а не 55 !
console.log(objC.c);          // 32
```

Сначала просматриваются все собственные свойства объекта, если свойство не найдено, то поиск свойства осуществляется в цепочке прототипов. В нашем примере внутри объекта `objC` мы создали свойство `b`, которое переопределяет одноименное свойство из объекта `objB`. Поэтому свойство `b` будет найдено в текущем объекте и поиск будет прекращен. В итоге мы получим значение 22.

Все объекты по умолчанию наследуют свойства класса `Object`. Поэтому, создав пустой объект, мы можем получить доступ, например, к методу `toString()`, определенному в классе `Object`. Пример:

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

Если свойству `__proto__` присвоить значение `null`, то объект не будет наследником класса `Object`:

```
let obj = {  
  a: 1,  
  __proto__: null  
};  
console.log( obj.a );           // 1  
console.log( obj.toString );    // undefined
```

Создать объект без наследования свойств класса `Object` позволяет статический метод `create(null)` из класса `Object`:

```
let obj = Object.create(null);  
obj.a = 1;  
console.log( obj.a );           // 1  
console.log( obj.toString );    // undefined
```

Свойство `__proto__` считается устаревшим, хотя и поддерживается Node.js и веб-браузерами. Теперь его рекомендуется заменять статическими методами `setPrototypeOf()` и `getPrototypeOf()` из класса `Object`:

```
<Объект>   = setPrototypeOf(<Объект>, <Прототип>)  
<Прототип> = getPrototypeOf(<Объект>)
```

Пример:

```
let objA = { a: 10 };  
let objB = { b: 55 };  
Object.setPrototypeOf(objB, objA);  
console.log(objB.a);           // 10  
console.log(objB.b);           // 55  
console.log( Object.getPrototypeOf(objB) ); // { a: 10 }
```

Пример создания объекта без наследования свойств класса `Object`:

```
let obj = Object.setPrototypeOf({}, null);  
obj.a = 1;  
console.log( obj.a );           // 1  
console.log( obj.toString );    // undefined
```

Метод `objA.isPrototypeOf(objB)` из класса `Object` возвращает значение `true`, если объект `objA` существует в цепочке прототипов объекта `objB`:

```
let objA = { a: 10 };  
let objB = { b: 55 };  
let objC = {};  
Object.setPrototypeOf(objB, objA);  
console.log( objA.isPrototypeOf(objB) ); // true  
console.log( objA.isPrototypeOf(objC) ); // false
```

```
Object.setPrototypeOf(objC, objB);
console.log( objA.isPrototypeOf(objC) ); // true
```

11.1.8. Атрибуты свойств и метод *Object.create()*

Создать объект можно с помощью статического метода `create()` из класса `Object`:

```
<Объект> = Object.create(<Прототип>[, <Опции>])
```

Если в первом параметре указать значение `null`, то будет создан объект без наследования свойств класса `Object`:

```
let obj = Object.create(null);
obj.a = 1;
console.log( obj.a );           // 1
console.log( obj.toString );    // undefined
```

Чтобы получить обычный объект, в первом параметре нужно указать значение `Object.prototype`:

```
let obj = Object.create(Object.prototype);
obj.a = 1;
console.log( obj.a );           // 1
console.log( obj.toString );    // [Function: toString]
```

Для наследования свойств какого-либо объекта в первом параметре следует указать этот объект:

```
let objA = Object.create(Object.prototype);
objA.a = 10;
let objB = Object.create(objA);
objB.b = 55;
console.log(objB.a);             // 10
console.log(objB.b);             // 55
```

Если в первом параметре указать значение, отличное от объекта и `null`, будет сгенерировано исключение:

```
let obj = Object.create(undefined);
// TypeError: Object prototype may only be an
// Object or null: undefined
```

Второй параметр позволяет сразу добавить свойства в созданный объект. В качестве значения указывается объект формата:

```
{
  <Свойство 1>: <Объект с настройками>[,
    ...,
  <Свойство N>: <Объект с настройками>]
}
```

Пример:

```
let objA = Object.create(Object.prototype, {
  'a': { value: 10 }
});
```

```
let objB = Object.create(objA, {  
  'b': { value: 55 },  
  'c': { value: 4 }  
});  
console.log(objB.a); // 10  
console.log(objB.b); // 55  
console.log(objB.c); // 4
```

Если мы сейчас попробуем вывести объекты в консоль, то получим пустые объекты, т. к. свойства не являются перечислимыми:

```
console.log(objA); // {}  
console.log(objB); // {}  
console.log( objA.propertyIsEnumerable('a') ); // false  
console.log( objB.propertyIsEnumerable('b') ); // false  
console.log( objB.propertyIsEnumerable('c') ); // false
```

Если попробуем изменить значение свойства, у нас ничего не получится, поскольку свойства не являются записываемыми:

```
objB.c = 17;  
console.log(objB.c); // 4, а не 17 ! в обычном режиме
```

Если попробуем удалить свойства, у нас опять ничего не получится, т. к. свойства не являются настраиваемыми:

```
console.log(delete objB.c); // false  
console.log(objB.c); // 4
```

В обычном режиме никаких ошибок не выводится. Однако при использовании строгого режима (включается добавлением инструкции 'use strict'; в начало файла) последние два примера приведут к исключениям:

```
TypeError: Cannot assign to read only property 'c' of object '#<Object>'  
TypeError: Cannot delete property 'c' of #<Object>
```

По умолчанию при создании объекта с помощью метода `create()` все атрибуты свойств имеют значение `false`, тогда как при создании объекта с помощью фигурных скобок все атрибуты имеют значение `true`. Для управления атрибутами нужно указать следующие опции в параметре <Объект с настройками>:

- ☐ `value` — задает значение свойству. Значение по умолчанию: `undefined`. Опцию нельзя указывать одновременно с опциями `set` и `get`;
- ☐ `writable` — если `true`, свойство станет доступным и для чтения, и для записи, если `false` — только для чтения. Значение по умолчанию: `false`. Опцию нельзя указывать одновременно с опциями `set` и `get`;
- ☐ `configurable` — если `true`, свойство станет *настраиваемым* (его можно будет удалить с помощью оператора `delete` и менять в дальнейшем), если `false` — не будет настраиваемым. Значение по умолчанию: `false`;
- ☐ `enumerable` — если `true`, свойство станет *перечислимым* (будет обрабатываться циклом `for...in`), если `false` — не будет перечислимым. Значение по умолчанию: `false`;

- ❑ `set` — задает функцию (setter) для изменения значения. Значение по умолчанию: `undefined`;
- ❑ `get` — задает функцию (getter) для получения значения. Значение по умолчанию: `undefined`.

В листинге 11.2 приведен пример указания опций во втором параметре метода `create()`.

**Листинг 11.2. Указание опций создаваемых свойств
во втором параметре метода `create()`**

```
let objA = Object.create(Object.prototype, {
  'a': {
    value: 10,
    writable: true,
    configurable: true,
    enumerable: true
  }
});
let objB = Object.create(objA, {
  'b': { value: 55, enumerable: true },
  'c': {
    enumerable: true,
    set: function(value) {
      this._c = value;
    },
    get: function() {
      return this._c;
    }
  },
  '_c': { writable: true }
});
console.log(objB.a); // 10
console.log(objB.b); // 55
console.log(objB._c); // undefined
objB.c = 4;
console.log(objB.c); // 4
console.log(objB._c); // 4
console.log(objA); // { a: 10 }
console.log(objB); // { b: 55, c: [Getter/Setter] }
```

11.1.9. Получение значений атрибутов свойств

Метод `propertyIsEnumerable(<Имя свойства>)` из класса `Object` возвращает значение `true`, если указанное свойство является перечислимым (обрабатывается циклом `for...in`), и `false` — в противном случае:

```
let obj = Object.create(Object.prototype, {
  'a': {
    value: 10,
    enumerable: true
  },
  'b': { value: 22 }
});
console.log( obj.propertyIsEnumerable('a') ); // true
console.log( obj.propertyIsEnumerable('b') ); // false
console.log( obj ); // { a: 10 }
```

Получить объект (дескриптор) со всеми атрибутами собственного свойства объекта `<Объект>` позволяет статический метод `getOwnPropertyDescriptor(<Объект>, <Имя свойства>)` из класса `Object`. Если свойство не найдено, метод вернет значение `undefined`. Пример использования метода `getOwnPropertyDescriptor()` приведен в листинге 11.3.

Листинг 11.3. Использование метода `getOwnPropertyDescriptor()`

```
let obj = Object.create(Object.prototype, {
  'a': {
    value: 10,
    writable: true,
    configurable: true,
    enumerable: true
  },
  'b': { value: 22 }
});
let d = Object.getOwnPropertyDescriptor(obj, 'a');
console.log(d);
// { value: 10, writable: true, enumerable: true,
//   configurable: true }
d = Object.getOwnPropertyDescriptor(obj, 'b');
console.log(d);
// { value: 22, writable: false, enumerable: false,
//   configurable: false }
d = Object.getOwnPropertyDescriptor(obj, 'c');
console.log(d); // undefined
```

Статический метод `getOwnPropertyDescriptors(<Объект>)` из класса `Object` возвращает объект со всеми собственными свойствами объекта `<Объект>` и их атрибутами. Если объект не содержит собственных свойств, метод вернет пустой объект. Пример применения метода `getOwnPropertyDescriptors()` приведен в листинге 11.4.

Листинг 11.4. Применение метода `getOwnPropertyDescriptors()`

```
let obj = Object.create(Object.prototype, {
  'a': {
    value: 10,
```

```

        writable: true,
        configurable: true,
        enumerable: true
    },
    'b': { value: 22 }
});
let o = Object.getOwnPropertyDescriptors(obj);
console.log(o);
/*
{
  a: { value: 10, writable: true, enumerable: true,
      configurable: true },
  b: {
    value: 22,
    writable: false,
    enumerable: false,
    configurable: false
  }
} */
obj = {};
o = Object.getOwnPropertyDescriptors(obj);
console.log(o); // {}

```

11.1.10. Методы *defineProperty()* и *defineProperties()*

Для управления атрибутами свойства после создания объекта можно воспользоваться статическим методом `defineProperty(<Объект>, <Имя свойства>, <Настройки>)` из класса `Object`. Метод возвращает ссылку на объект. Использование метода `defineProperty()` иллюстрирует листинг 11.5.

Листинг 11.5. Использование метода `defineProperty()`

```

let obj = Object.create(Object.prototype, {
  'a': {
    value: 10,
    writable: true,
    enumerable: true,
    configurable: true
  }
});
// Добавление нового свойства
Object.defineProperty(obj, 'b', {
  value: 22,
  writable: true,
  configurable: true,
  enumerable: true
});

```

```
console.log( obj.b ); // 22
// Изменение атрибутов существующего свойства
console.log( obj.propertyIsEnumerable('a') ); // true
Object.defineProperty(obj, 'a', { enumerable: false });
console.log( obj.propertyIsEnumerable('a') ); // false
// Setter и getter
Object.defineProperty(obj, 'c', {
  set: function(value) {
    this._c = value;
  },
  get: function() {
    return this._c;
  }
});
obj.c = 34;
console.log( obj.c ); // 34
```

Как видно из примера, метод `defineProperty()` позволяет добавить новое свойство, `setter` и `getter`. При добавлении свойства атрибуты `writable`, `configurable` и `enumerable` по умолчанию получают значение `false`. Метод позволяет также изменить значения атрибутов существующего свойства. Следует учитывать, что попытка изменить свойство при опции `configurable`, имеющей значение `false`, приведет к исключению. Пример:

```
let obj = Object.create(Object.prototype, {
  'a': {
    value: 10,
    writable: true,
    configurable: false,
    enumerable: true
  }
});
Object.defineProperty(obj, 'a', { enumerable: false });
// TypeError: Cannot redefine property: a
```

Однако в этом случае можно изменить значение атрибута `writable` на `false`.

```
Object.defineProperty(obj, 'a', { writable: false }); // OK
```

Статический метод `defineProperty()` из класса `Reflect` (см. *разд. 11.11*) в этом случае вместо генерации исключения просто вернет значение `false`:

```
console.log(
  Reflect.defineProperty(obj, 'a', { enumerable: false })
); // false
```

Добавить или изменить сразу несколько свойств можно статическим методом `defineProperties(<Объект>, <Опции>)` из класса `Object`. Параметр `<Опции>` аналогичен одноименному параметру метода `create()`. Метод `defineProperties()` возвра-

щает ссылку на объект. Использование этого метода показано в коде из листинга 11.6.

Листинг 11.6. Использование метода `defineProperties()`

```
let obj = Object.create(Object.prototype, {
  'a': {
    value: 10,
    writable: true,
    enumerable: true,
    configurable: true
  }
});
Object.defineProperties(obj, {
  // Добавление нового свойства
  'b': {
    value: 22,
    writable: true,
    configurable: true,
    enumerable: true
  },
  // Изменение атрибутов существующего свойства
  'a': { enumerable: false },
  // Setter и getter
  'c': {
    set: function(value) {
      this._c = value;
    },
    get: function() {
      return this._c;
    }
  }
});
console.log( obj.b ); // 22
console.log( obj.propertyIsEnumerable('a') ); // false
obj.c = 34;
console.log( obj.c ); // 34
```

При добавлении свойства атрибуты `writable`, `configurable` и `enumerable` по умолчанию получают значение `false`. Попытка изменить свойство при опции `configurable`, имеющей значение `false`, приведет к исключению. Однако в этом случае можно изменить значение атрибута `writable` на `false`.

11.1.11. Ограничение доступа к объекту

Статический метод `freeze(<Объект>)` из класса `Object` позволяет сделать объект доступным только для чтения ("заморозить"). Нельзя будет изменить значения

свойств и их атрибутов, добавить или удалить свойство. Однако можно менять значения свойств вложенных объектов при условии, что они не "заморожены" отдельно. Попытка изменения объекта в строгом режиме приведет к исключению. Метод возвращает ссылку на объект.

Проверить, "заморожен" ли объект, позволяет статический метод `isFrozen(<Объект>)` из класса `Object`. Метод возвращает значение `true`, если объект "заморожен", и `false` — в противном случае.

Пример "заморозки" объекта приведен в листинге 11.7.

Листинг 11.7. "Заморозка" объекта

```
'use strict';
let obj = {
  a: 10,
  set c(value) {
    this._c = value;
  },
  get c() {
    return this._c;
  },
  _c: 0
};

console.log(obj); // { a: 10, c: [Getter/Setter], _c: 0 }
console.log( Object.isFrozen(obj) ); // false
Object.freeze(obj);
console.log( Object.isFrozen(obj) ); // true
// obj.a = 22;           // Нельзя изменить значение
// В строгом режиме получим исключение
// TypeError: Cannot assign to read only property 'a'
// of object '#<Object>'
console.log(obj); // { a: 10, c: [Getter/Setter], _c: 0 }
// delete obj.a;         // Нельзя удалить свойство
// В строгом режиме получим исключение
// TypeError: Cannot delete property 'a' of #<Object>
console.log(obj); // { a: 10, c: [Getter/Setter], _c: 0 }
// Object.defineProperty(obj, 'b', { value: 14});
// TypeError: Cannot define property b, object is not extensible
// Нельзя добавить свойство или изменить атрибуты
console.log(obj); // { a: 10, c: [Getter/Setter], _c: 0 }
// obj.c = 10;           // Setter не изменяет значение
// В строгом режиме получим исключение
// TypeError: Cannot assign to read only property '_c'
// of object '#<Object>'
console.log(obj.c); // 0
console.log(obj); // { a: 10, c: [Getter/Setter], _c: 0 }
```

```
// Можно изменять вложенные объекты
obj = Object.freeze({ a: { b: 5 }});
obj.a.b = 3;
console.log(obj); // { a: { b: 3 } }
```

Статический метод `seal(<Объект>)` из класса `Object` запечатывает объект. Нельзя будет добавить новое свойство и удалить существующее, т. к. метод делает все свойства ненастраиваемыми (атрибут `configurable` получит значение `false`). При этом можно менять значения свойств. `Setter` может изменить значение существующего свойства, но не может добавить новое свойство. Метод возвращает ссылку на объект.

Проверить, является ли объект запечатанным, позволяет статический метод `isSealed(<Объект>)` из класса `Object`. Метод возвращает значение `true`, если объект запечатан, и `false` — в противном случае.

Пример запечатывания объекта приведен в листинге 11.8.

Листинг 11.8. Запечатывание объекта

```
'use strict';
let obj = {
  a: 10,
  set c(value) {
    this._c = value;
  },
  get c() {
    return this._c;
  },
  _c: 0
};
console.log(obj); // { a: 10, c: [Getter/Setter], _c: 0 }
console.log( Object.isSealed(obj) ); // false
Object.seal(obj);
console.log( Object.isSealed(obj) ); // true
// delete obj.a; // Нельзя удалить свойство
// В строгом режиме получим исключение
// TypeError: Cannot delete property 'a' of #<Object>
console.log(obj); // { a: 10, c: [Getter/Setter], _c: 0 }
// Object.defineProperty(obj, 'b', { value: 14});
// TypeError: Cannot define property b, object is not extensible
// Нельзя добавить свойство
console.log(obj); // { a: 10, c: [Getter/Setter], _c: 0 }
obj.a = 22; // Можно изменить значение
console.log(obj); // { a: 22, c: [Getter/Setter], _c: 0 }
obj.c = 10;
// Setter изменяет значение, только если не нужно добавлять свойство
console.log(obj.c); // 10
console.log(obj); // { a: 22, c: [Getter/Setter], _c: 10 }
```

Статический метод `preventExtensions(<Объект>)` из класса `Object` запрещает расширение объекта. Нельзя будет добавить новое свойство. При этом можно менять значения свойств и удалять их. Метод возвращает ссылку на объект.

Проверить, можно ли расширять объект путем добавления новых свойств, позволяет статический метод `isExtensible(<Объект>)` из класса `Object`. Метод возвращает значение `true`, если объект является расширяемым, и `false` — в противном случае.

Пример запрета на расширение объекта показан в листинге 11.9.

Листинг 11.9. Запрет расширения объекта

```
'use strict';
let obj = { a: 10 };
console.log(obj);    // { a: 10 }
console.log( Object.isExtensible(obj) ); // true
Object.preventExtensions(obj);
console.log( Object.isExtensible(obj) ); // false
obj.a = 22;          // Можно изменить значение
console.log(obj);    // { a: 22 }
delete obj.a;        // Можно удалить свойство
console.log(obj);    // {}
// Object.defineProperty(obj, 'b', { value: 14});
// TypeError: Cannot define property b, object is not extensible
// Нельзя добавить свойство
// obj.c = 10;
// В строгом режиме получим исключение
// TypeError: Cannot add property c, object is not extensible
console.log(obj);    // {}
```

При использовании методов `freeze()` и `seal()` объект также становится нерасширяемым:

```
let obj1 = { a: 10 }, obj2 = { b: 3 };
Object.freeze(obj1);
console.log( Object.isExtensible(obj1) ); // false
Object.seal(obj2);
console.log( Object.isExtensible(obj2) ); // false
```

11.1.12. Получение списка имен свойств

Статический метод `keys(<Объект>)` из класса `Object` возвращает массив с именами собственных перечислимых свойств объекта. Цепочка прототипов не просматривается. Пример:

```
let objA = Object.create(Object.prototype, {
  'a': { value: 10, enumerable: true },
  'c': { value: 22, enumerable: false }
});
```

```
let objB = Object.create(objA, {
  'b': { value: 55, enumerable: true }
});
console.log( Object.keys(objA) ); // [ 'a' ]
console.log( Object.keys(objB) ); // [ 'b' ]
```

В этом примере мы не получили свойство `c`, поскольку оно не является перечислимым. Если нужно дополнительно получить все собственные свойства, следует воспользоваться статическим методом `getOwnPropertyNames(<Объект>)` из класса `Object`. Пример:

```
console.log( Object.getOwnPropertyNames(objA) ); // [ 'a', 'c' ]
console.log( Object.getOwnPropertyNames(objB) ); // [ 'b' ]
```

Статический метод `values(<Объект>)` из класса `Object` возвращает массив со значениями собственных перечислимых свойств объекта. Цепочка прототипов не просматривается. Пример:

```
console.log( Object.values(objA) ); // [ 10 ]
console.log( Object.values(objB) ); // [ 55 ]
```

С помощью статического метода `entries(<Объект>)` из класса `Object` можно получить массив, каждый элемент которого содержит массив, состоящий из имени и значения одного из собственных перечислимых свойств:

```
console.log( Object.entries(objA) ); // [ [ 'a', 10 ] ]
console.log( Object.entries(objB) ); // [ [ 'b', 55 ] ]
```

Напишем функцию, которая будет возвращать массив из двух массивов: в первом массиве будут храниться имена всех собственных свойств объекта, а во втором — имена всех свойств в его прототипе (без просмотра всей цепочки):

```
function getProps(obj) {
  if (typeof obj !== 'object' || obj === null)
    return [ [], [] ];
  let arrOwnProps = Object.getOwnPropertyNames(obj);
  arrOwnProps.sort();
  if (obj.__proto__ === null)
    return [ arrOwnProps, [] ];
  let arrProtoProps = Object.getOwnPropertyNames(obj.__proto__);
  arrProtoProps.sort();
  return [ arrOwnProps, arrProtoProps ]
}
let obj = { a: 0 };
console.log( getProps(obj) );
```

Результат:

```
[
  [ 'a' ],
  [
    '__defineGetter__',

```

```

    '__defineSetter__',
    '__lookupGetter__',
    '__lookupSetter__',
    '__proto__',
    'constructor',
    'hasOwnProperty',
    'isPrototypeOf',
    'propertyIsEnumerable',
    'toLocaleString',
    'toString',
    'valueOf'
  ]
]

```

Статический метод `ownKeys(<Объект>)` из класса `Reflect` возвращает массив с именами собственных свойств, включая символные:

```

let obj = Object.create(Object.prototype, {
  'a': { value: 3, enumerable: true },
  'b': { value: 6, enumerable: false },
  [Symbol.for('c')]: { value: 9, enumerable: true }
});
console.log( Reflect.ownKeys(obj) );
// [ 'a', 'b', Symbol(c) ]

```

11.1.13. Перебор свойств объекта

Получить имена всех перечислимых свойств объекта позволяет цикл `for...in`:

```

let objA = Object.create(Object.prototype, {
  'a': { value: 10, enumerable: true },
  'c': { value: 22, enumerable: false }
});
let objB = Object.create(objA, {
  'b': { value: 55, enumerable: true }
});
for (let prop in objB) {
  // Переменной prop на каждой итерации присваивается
  // имя перечислимого свойства объекта
  console.log(`${prop} = ${objB[prop]}`);
}
// b = 55
// a = 10

```

Перебрать можно только перечислимые свойства (для которых метод `propertyIsEnumerable()` из класса `Object` вернул значение `true`). Для свойств встроенных объектов метод возвращает `false`, поэтому перебрать свойства этих объектов с помощью цикла `for...in` нельзя. Пример:

```

console.log( objB.propertyIsEnumerable('b') );           // true
console.log( objA.propertyIsEnumerable('a') );           // true
console.log( objA.propertyIsEnumerable('c') );           // false
console.log( objB.toString() );                           // [object Object]
console.log( objB.propertyIsEnumerable('toString') );    // false
let x = new Number(10);
console.log( x.propertyIsEnumerable('toFixed') );        // false

```

Цикл `for...in` имеет доступ не только к свойствам текущего объекта, но и к унаследованным свойствам в цепочке прототипов. Если эти унаследованные свойства не нужны, их следует отфильтровать с помощью метода `hasOwnProperty()`. Пример:

```

for (let prop in objB) {
  if ( objB.hasOwnProperty(prop) ) {
    console.log(`YES: ${prop}`);
  }
  else {
    console.log(`NO: ${prop}`);
  }
}
// YES: b
// NO: a

```

11.1.14. Проверка существования свойств и методов

Оператор `in` позволяет проверить существование свойства (включая унаследованные в цепочке прототипов) у объекта. Если свойство существует, возвращается значение `true`. Пример:

```

let objA = { a: 10 };
let objB = { b: 55 };
objB.__proto__ = objA;
console.log('a' in objA);           // true
console.log('b' in objA);           // false
console.log('a' in objB);           // true
console.log('b' in objB);           // true
console.log('toString' in objB);    // true
let x = new Number(5);
console.log('toFixed' in x);        // true

```

Проверить наличие собственного (не унаследованного в цепочке прототипов) свойства позволяет метод `hasOwnProperty(<Имя свойства>)`:

```

let objA = { a: 10 };
let objB = { b: 55 };
objB.__proto__ = objA;
console.log('a' in objB);           // true
console.log( objB.hasOwnProperty('a') ); // false
console.log( objB.hasOwnProperty('b') ); // true

```

Если имя метода указать в условии без круглых скобок, то это позволит проверить наличие метода:

```
let x = new Number(5);
if (x.toFixed) console.log('Метод определен');
else           console.log('Нет');
```

ОБРАТИТЕ ВНИМАНИЕ

Проверять таким образом наличие свойства нельзя, т. к. значение 0 будет интерпретировано как false.

11.1.15. Свойства с типом *symbol*

Свойства объекта могут иметь тип *symbol* (символ), который описывает уникальное и неизменяемое значение. Создать переменную с типом *symbol* можно так:

```
<Переменная> = Symbol([<Описание>])    // Локальный символ
<Переменная> = Symbol.for(<Имя>)        // Глобальный символ
```

Листинг 11.10 иллюстрирует создание и использование свойств символьного типа.

Листинг 11.10. Свойства типа *symbol*

```
// Локальные символы
let sym1 = Symbol();
let sym2 = Symbol('sym2');
console.log(sym1);           // Symbol()
console.log(sym2);           // Symbol(sym2)
let sym3 = Symbol('sym2');
console.log(sym3);           // Symbol(sym2)
console.log(sym2 === sym3);  // false
// Глобальные символы
let sym4 = Symbol.for('sym4');
console.log(sym4);           // Symbol(sym4)
let sym5 = Symbol.for('sym4');
console.log(sym5);           // Symbol(sym4)
console.log(sym4 === sym5);  // true
```

При указании одинакового описания в первом случае создается новый уникальный символ, поэтому символы не равны друг другу. При использовании метода *for()* сначала проверяется наличие символа с указанным в качестве параметра именем в глобальном реестре. Если символ найден, он возвращается методом *for()*, в противном случае создается новый глобальный символ. Поэтому во втором случае мы получили равенство значений переменных.

Получить имя символа из глобального реестра позволяет статический метод *keyFor(<Символ>)* из класса *Symbol*. Если символа нет в реестре, метод вернет значение *undefined*.

ОБРАТИТЕ ВНИМАНИЕ

Локальные символы не добавляются в глобальный реестр.

Пример:

```
console.log( Symbol.keyFor(sym1) ); // undefined
console.log( Symbol.keyFor(sym2) ); // undefined
console.log( Symbol.keyFor(sym4) ); // sym4
console.log( Symbol.keyFor(sym5) ); // sym4
```

При добавлении свойства с типом `symbol` в объект его значение указывается внутри квадратных скобок:

```
let sym1 = Symbol();
let sym2 = Symbol.for('sym2');
let obj = {
  [sym1]: 20
};
obj[sym2] = 3;
console.log( obj );
// { [Symbol()]: 20, [Symbol(sym2)]: 3 }
```

Чтобы получить значение свойства с типом `symbol`, нужно иметь символ, ранее сохраненный в переменной, или воспользоваться методом `for()` для получения символа по имени:

```
console.log( obj[sym1] ); // 20
console.log( obj[Symbol.for('sym2')] ); // 3
```

Следует учитывать, что свойства с типом `symbol` нельзя получить с помощью методов `keys()`, `values()`, `entries()` и `getOwnPropertyNames()`, они также недоступны для перебора с помощью цикла `for...in`:

```
console.log( Object.keys(obj) ); // []
console.log( Object.values(obj) ); // []
console.log( Object.entries(obj) ); // []
console.log( Object.getOwnPropertyNames(obj) ); // []
for (let prop in obj) {
  console.log(`${prop}`);
} // Ничего не выведет
```

Для получения свойств с типом `symbol` следует воспользоваться статическим методом `getOwnPropertySymbols(<Объект>)` из класса `Object`, который возвращает массив с такими свойствами:

```
console.log( Object.getOwnPropertySymbols(obj) );
// [ Symbol(), Symbol(sym2) ]
```

11.1.16. Сравнение объектов

При создании объекта в переменной сохраняется ссылка на объект, а не сам объект. Поэтому сравнение двух пустых объектов всегда возвращает `false`. Пример:

```
console.log( {} == {} ); // false
console.log( {} === {} ); // false
```

Сравнение объектов вернет `true` только в том случае, если две переменные ссылаются на один и тот же объект:

```
let obj1 = { a: 10 }, obj2 = { a: 10 };
console.log( obj1 == obj2 ); // false
console.log( obj1 === obj2 ); // false
let obj3 = obj1;
console.log( obj1 == obj3 ); // true
console.log( obj1 === obj3 ); // true
```

Выполнить сравнение объектов также позволяет статический метод `is(<Значение 1>, <Значение 2>)` из класса `Object`. Он возвращает значение `true`, если переменные содержат ссылку на один и тот же объект, и `false` — в противном случае. Пример:

```
let obj1 = { a: 10 }, obj2 = { a: 10 }, obj3 = obj1;
console.log( Object.is(obj1, obj2) ); // false
console.log( Object.is(obj1, obj3) ); // true
```

11.1.17. Создание копии объекта

Важно учитывать, что простое присваивание копирует лишь ссылку на объект, а не сам объект. В результате обе переменные будут ссылаться на один и тот же объект. Пример:

```
let obj1 = { a: 10 };
let obj2 = obj1;
obj2.a = 22;
console.log( obj1 ); // { a: 22 }
console.log( obj2 ); // { a: 22 }
```

Для создания поверхностной копии объекта можно воспользоваться оператором `SPREAD`. При этом копируются только собственные перечислимые свойства объекта. Пример поверхностного копирования объекта приведен в листинге 11.11.

Листинг 11.11. Создание поверхностной копии объекта

```
let obj1 = Object.create(Object.prototype, {
  'a': {
    value: 10,
    writable: true,
    configurable: true,
    enumerable: true
  },
  'b': { value: 3, enumerable: false }
});
let obj2 = { ...obj1 };
obj2.a = 22;
```

```
console.log( obj1 ); // { a: 10 }
console.log( obj2 ); // { a: 22 }
// Копируются только собственные перечислимые свойства
console.log( Object.getOwnPropertyNames(obj2) ); // [ 'a' ]
```

Свойство `b` из этого примера не является перечислимым, поэтому оно не было скопировано. Вложенные объекты не копируются — копируются лишь ссылки на эти объекты, поэтому мы получим поверхностную, а не глубокую копию объекта. Пример:

```
let obj = { a: { b: 10 } };
let objCopy = { ...obj };
objCopy.a.b = 22;
console.log( obj ); // { a: { b: 22 } }
console.log( objCopy ); // { a: { b: 22 } }
```

Для копирования собственных перечислимых свойств также можно воспользоваться статическим методом `assign()` из класса `Object`. В первом параметре указываем пустой объект, а во втором параметре передаем ссылку на копируемый объект. Метод возвращает ссылку на объект из первого параметра. Использование метода `assign()` показано в листинге 11.12.

Листинг 11.12. Копирование собственных перечислимых свойств объекта вызовом метода `assign()`

```
let obj1 = Object.create(Object.prototype, {
  'a': {
    value: 10,
    writable: true,
    configurable: true,
    enumerable: true
  },
  'b': { value: 3, enumerable: false }
});
let obj2 = Object.assign({}, obj1);
obj2.a = 22;
console.log( obj1 ); // { a: 10 }
console.log( obj2 ); // { a: 22 }
// Копируются только собственные перечислимые свойства
console.log( Object.getOwnPropertyNames(obj2) ); // [ 'a' ]
```

Для копирования всех собственных свойств можно воспользоваться кодом из листинга 11.13.

Листинг 11.13. Копирование всех собственных свойств объекта

```
let obj1 = Object.create(Object.prototype, {
  'a': {
    value: 10,
```

```
writable: true,
configurable: true,
enumerable: true
},
'b': { value: 3, enumerable: false }
});
let obj2 = Object.create(
    Object.getPrototypeOf(obj1),
    Object.getOwnPropertyDescriptors(obj1) );
obj2.a = 22;
console.log( obj1 ); // { a: 10 }
console.log( obj2 ); // { a: 22 }
// Копируются все собственные свойства
console.log( Object.getOwnPropertyNames(obj2) ); // [ 'a', 'b' ]
```

11.1.18. Объединение объектов

Для объединения объектов можно воспользоваться оператором `SPREAD`. При этом копируются только собственные перечислимые свойства. Пример объединения объектов таким образом показан в листинге 11.14.

Листинг 11.14. Объединение объектов с помощью оператора `SPREAD`

```
let obj1 = { a: 15, c: 6 };
let obj2 = Object.create(Object.prototype, {
  'a': {
    value: 10,
    writable: true,
    configurable: true,
    enumerable: true
  },
  'b': { value: 3, enumerable: false }
});
let obj3 = { ...obj1, ...obj2 }
console.log( obj3 ); // { a: 10, c: 6 }
// Копируются только собственные перечислимые свойства
console.log( Object.getOwnPropertyNames(obj3) ); // [ 'a', 'c' ]
```

Если свойства имеют одинаковое имя, то значение из первого объекта будет заменено значением из второго объекта. Поэтому свойство `a` из данного примера получило значение 10. Эту особенность удобно использовать для объединения объекта с настройками по умолчанию и объекта с настройками, заданными программистом. Пример:

```
let defaultObj = { focus: true, show: true };
let userObj = { show: false };
console.log( { ...defaultObj, ...userObj } );
// { focus: true, show: false }
```

Для объединения нескольких объектов с объектом из первого параметра можно воспользоваться статическим методом `assign(<Объект>, ...<Объекты>)` из класса `Object`. Метод возвращает ссылку на объект из первого параметра. Пример объединения объектов вызовом метода `assign()` приведен в листинге 11.15.

Листинг 11.15. Объединение объектов с помощью метода `assign()`

```
let obj1 = { a: 15, c: 6 };
let obj2 = Object.create(Object.prototype, {
  'a': {
    value: 10,
    writable: true,
    configurable: true,
    enumerable: true
  },
  'b': { value: 3, enumerable: false }
});
let obj3 = Object.assign(obj1, obj2);
console.log( obj3 );           // { a: 10, c: 6 }
// Изменяется объект из первого параметра
console.log( obj1 );           // { a: 10, c: 6 }
console.log( obj3 === obj1 ); // true
// Копируются только собственные перечислимые свойства
console.log( Object.getOwnPropertyNames(obj3) ); // [ 'a', 'c' ]
```

Если не нужно изменять существующий объект, то в первом параметре следует указать пустой объект:

```
let obj4 = Object.assign({}, obj1, obj2);
console.log( obj4 );           // { a: 10, c: 6 }
console.log( obj4 === obj1 ); // false
```

11.1.19. Удаление свойства

Удалить свойство позволяет оператор `delete`:

```
let obj = { a: 1, b: 2, c: 3 };
delete obj.b;
delete obj['c'];
console.log(obj);           // { a: 1 }
```

Если у свойства были указаны `getter` и `setter`, оба будут удалены:

```
let car = {
  set model(name) {
    this._model = name;
  },

```

```
    get model() {
        return this._model;
    }
};
console.log(car);    // { model: [Getter/Setter] }
delete car.model;
console.log(car);    // {}
```

Оператор `delete` возвращает значение `true`, если свойство успешно удалено, и `false` — в противном случае:

```
let obj = { a: 1, b: 2, c: 3 };
console.log(delete obj.b);    // true
```

ОБРАТИТЕ ВНИМАНИЕ

Нельзя удалить свойства встроенных объектов. В обычном режиме оператор `delete` вернет значение `false`, а в строгом режиме будет сгенерировано исключение.

Пример:

```
console.log(delete Math.PI);
// false в обычном режиме
// В строгом режиме:
// TypeError: Cannot delete property 'PI' of #<Object>
```

Удалить свойство также позволяет статический метод `deleteProperty(<Объект>, <Свойство>)` из класса `Reflect`. Метод возвращает значение `true`, если свойство было успешно удалено из объекта, и `false` — в противном случае. Если свойство не существует, метод вернет значение `true`. Пример:

```
let obj = { a: 3 };
console.log( Reflect.deleteProperty(obj, 'a') ); // true
console.log( Reflect.deleteProperty(obj, 'b') ); // true
console.log( obj );                               // {}
```

В строгом режиме попытка удалить свойство с атрибутом `configurable`, равным `false`, с помощью оператора `delete` приведет к исключению, тогда как метод `deleteProperty()` просто вернет `false`:

```
'use strict';
let obj = Object.freeze({ a: 3 });
console.log( Reflect.deleteProperty(obj, 'a') ); // false
console.log( obj );                               // { a: 3 }
// delete obj.a;
// TypeError: Cannot delete property 'a' of #<Object>
let d = Object.getOwnPropertyDescriptor(obj, 'a');
console.log(d);
// { value: 3, writable: false, enumerable: true,
//   configurable: false }
```

11.2. Создание объекта с помощью класса *Object*

Создать новый объект также можно с помощью класса `Object` (листинг 11.16).

Листинг 11.16. Создание объекта с помощью класса `Object`

```
let car = new Object();
console.log( car );           // {}
car._model = 'BAZ-2109';     // Сохранили строку
car._year = 2007;            // Сохранили число
car.getYear = function() {   // Сохранили ссылку на функцию
    return this._year;
};
// setter и getter
Object.defineProperty(car, 'model', {
    set: function(name) {
        this._model = name;
    },
    get: function() {
        return this._model;
    },
    enumerable: true,
    configurable: true
});
// Вывод значений
console.log( car._year );     // 2007
console.log( car.getYear() ); // 2007
// setter и getter
car.model = 'BMB';
console.log( car.model );     // BMB
console.log( car._model );    // BMB
console.log( car );
// { _model: 'BMB', _year: 2007, getYear: [Function],
//   model: [Getter/Setter] }
```

После создания объекта в переменной `car` сохраняется ссылка на него. Используя точечную нотацию, можно добавить произвольное свойство. Если в качестве значения свойства указать ссылку на функцию, то такое свойство становится методом объекта, внутри которого доступен указатель `this` на текущий объект.

Класс `Object` является базовым для всех других классов в JavaScript, включая пользовательские. Все классы наследуют метод `toString()`, который возвращает значение объекта в виде строки. Метод вызывается автоматически при использовании объекта в строковом контексте. Пример:

```
console.log( car.toString() ); // [object Object]
console.log( car + ' ' );      // [object Object]
```

Этот метод можно переопределить в пользовательском объекте:

```
let point = new Object();
point.x = 10;
point.y = 5;
point.toString = function() {
  return `x: ${this.x}, y: ${this.y}`;
};
console.log( point.toString() );      // x: 10, y: 5
```

При создании объекта с помощью фигурных скобок неявным образом вызывается конструктор класса `Object`. Таким образом эти две инструкции создания объектов эквивалентны:

```
let obj1 = { a: 1, b: 2 };
let obj2 = new Object({ a: 1, b: 2 });
console.log(obj1); // { a: 1, b: 2 }
console.log(obj2); // { a: 1, b: 2 }
```

Создать объект на основе другого объекта, поддерживающего итерации, позволяет статический метод `fromEntries(<Итератор>)` из класса `Object`. Каждый элемент такого объекта должен состоять из пары `ключ/значение` или из вложенного массива, содержащего ключ и значение. Пример:

```
let arr = [ ['a', 1], ['b', 2] ];
let obj = Object.fromEntries(arr);
console.log(obj); // { a: 1, b: 2 }
let map = new Map();
map.set('c', 10);
map.set('d', 20);
obj = Object.fromEntries(map);
console.log(obj); // { c: 10, d: 20 }
```

Выполнить обратную процедуру позволяет метод `entries()`:

```
let obj = { a: 1, b: 2 };
let arr = Object.entries(obj);
console.log(arr); // [ [ 'a', 1 ], [ 'b', 2 ] ]
```

11.3. Создание класса (старый стиль)

В предыдущих разделах мы создавали один объект. Однако часто нужно создать множество объектов, обладающих одинаковым набором свойств и методов, но имеющих разные значения свойств. В этом случае следует описать *класс*. Класс можно сравнить с чертежом, а объект — с изделием, произведенным по этому чертежу. Чертеж всегда один, а вот изделий может быть бесконечное множество.

В языке JavaScript существуют два стиля описания класса: старый и новый. При старом стиле в качестве конструктора класса выступает обычная функция. В новом стиле класс описывается с помощью инструкции `class`.

11.3.1. Функция в качестве конструктора класса

Если после ключевого слова `new` указана функция, то она становится *конструктором* класса, создающим новый объект этого класса. Конструктору можно передать начальные значения для свойств создаваемого объекта (листинг 11.17). Внутри конструктора доступен указатель `this` на текущий объект.

ОБРАТИТЕ ВНИМАНИЕ

Так как стрелочные функции не имеют своего указателя `this`, их нельзя использовать в качестве конструкторов.

Листинг 11.17. Функция в качестве конструктора класса

```
function Car(m, y) { // Конструктор класса
  if ( !new.target ) throw 'Функция вызвана без new';
  this.model = m;
  this.year = y;
  this.getModel = function() {
    return this.model;
  }
}

// Создание объектов
let car1 = new Car('BA3-2109', 2007);
let car2 = new Car('LADA Granta', 2014);
// Вывод значений
console.log( car1.year );           // 2007
console.log( car1.getModel() );    // BA3-2109
console.log( car2.year );           // 2014
console.log( car2.getModel() );    // LADA Granta
// Изменение значения не влияет на второй объект
car1.year = 2008;
console.log( car1 );
// Car { model: 'BA3-2109', year: 2008,
//   getModel: [Function] }
console.log( car2 );
// Car { model: 'LADA Granta', year: 2014,
//   getModel: [Function] }
```

Для создания объекта на основе написанного класса используется оператор `new`. После него указывается имя нужного класса и, в круглых скобках, параметры, передаваемые конструктору. Формат:

```
<Переменная> = new <Имя класса>([<Параметры>])
```

Если конструктор класса не принимает параметров, они не указываются (однако круглые скобки указать надо):

```
let x = new Number(10);
console.log(typeof x);           // object
```

По ошибке при создании объекта можно забыть указать оператор `new`. В этом случае функция будет просто вызвана без создания объекта. Чтобы этого избежать, следует проверить значение `new.target`. Если был использован оператор `new`, `new.target` будет содержать ссылку на конструктор, в противном случае — значение `undefined`. Пример вывода значения `new.target`:

```
function A() {  
    console.log( new.target );  
}  
new A();    // [Function: A]  
A();        // undefined
```

Для быстрой вставки инструкции с оператором `new` в редакторе VS Code предназначена следующая комбинация:

```
new =>  
const name = new type(arguments);
```

11.3.2. Прототипы

В листинге 11.17 мы определяли метод `getModel()` внутри конструктора:

```
function Car(m, y) { // Конструктор класса  
    this.model = m;  
    this.year = y;  
    this.getModel = function() {  
        return this.model;  
    }  
}
```

Подобное решение неэффективно, поскольку у каждого создаваемого объекта будет создан отдельный объект функции (относящийся к классу `Function`), представляющей этот метод, что приведет к неоправданному расходу памяти.

Использование прототипов позволяет определить метод вне конструктора. Каждый создаваемый объект наследует все свойства, которые имеются в прототипе. Таким образом, метод `getModel()` будет определен один раз, а отдельные объекты будут хранить ссылку на представляющий его объект, которая не займет много места в памяти.

Для добавления метода в прототип используется свойство `prototype` (листинг 11.18).

Листинг 11.18. Прототипы

```
function Car(m, y) {  
    this.model = m;  
    this.year = y;  
}
```

```

Car.prototype.getModel = function() {
    return this.model;
}
// Создание объектов
let obj1 = new Car('BA3-2109', 2007);
let obj2 = new Car('LADA Granta', 2014);
// Вывод значений
console.log(obj1.year);           // 2007
console.log(obj1.getModel());    // BA3-2109
console.log(obj2.year);           // 2014
console.log(obj2.getModel());    // LADA Granta

```

Как уже говорилось, свойства, определенные в прототипе класса, наследуются всеми объектами этого класса. Таким образом, метод `getModel()` доступен для перебора в цикле `for...in`, а также успешно проверяется на наличие с помощью оператора `in`. Тем не менее метод `hasOwnProperty()` позволяет определить, что заданный метод объекта является унаследованным. Пример:

```

console.log( 'getModel' in obj1 );           // true
console.log( obj1.hasOwnProperty('getModel') ); // false

```

Как вы уже знаете, все объекты имеют свойство `__proto__`, ссылающееся на прототип объекта. В свою очередь, все функции и классы содержат свойство `prototype`, которое также ссылается на прототип. Эти два свойства ссылаются на один и тот же прототип, если объект создан на основе функции или класса. Пример:

```

console.log(obj1.__proto__ === Car.prototype);           // true
console.log(obj2.__proto__ === Car.prototype);           // true
console.log(Object.getPrototypeOf(obj1) === Car.prototype); // true

```

Функции и классы являются объектами, поэтому они также содержат свойство `__proto__`, которое ссылается на прототип класса `Function`:

```

console.log(Car.__proto__ === Function.prototype);       // true
console.log(
    Function.prototype.__proto__ === Object.prototype);  // true

```

Почти все встроенные классы JavaScript (например, `String`, `Array`) имеют свойство `prototype`. С его помощью можно расширить возможности встроенных классов (например, добавить новый метод). В качестве примера добавим в класс `Array` метод `inArray()`. Он будет искать в массиве заданное значение и возвращать индекс первого вхождения. Если вхождение не найдено, метод вернет значение `-1`. Код метода:

```

Array.prototype.inArray = function(elem) {
    for (let i = 0, len = this.length; i < len; i++) {
        if (this[i] === elem) return i;
    }
    return -1;
}

```

```
let arr = [ 1, 2, 3, 4, 5, 1 ];
let pos = arr.indexOf(5);
if (pos !== -1) console.log(`Индекс элемента ${pos}`);
else console.log('Не найдено');
```

Если добавить метод в прототип класса `Object`, то метод будет доступен всем объектам:

```
Object.prototype.printHello = function() {
  console.log('Hello');
}
let arr = [ 1, 2, 3, 4, 5, 1 ];
arr.printHello();    // Hello
let obj = { a: 1 };
obj.printHello();    // Hello
let x = 4;
x.printHello();      // Hello
```

11.3.3. Свойство *constructor*

Свойство `constructor` содержит ссылку на функцию-конструктор, с помощью которой был создан объект. Если объект создавался с помощью фигурных скобок, то это свойство будет хранить ссылку на `Object`. Пример:

```
function Car(m, y) {
  this.model = m;
  this.year = y;
}
let obj1 = { a: 1 };
console.log(obj1.constructor === Object); // true
let obj2 = new Car('LADA Granta', 2014);
console.log(obj2.constructor === Car);    // true
```

11.3.4. Наследование

В разд. 11.1.7 мы уже рассматривали возможность расширения объекта за счет наследования свойств другого объекта. В этом случае базовый объект указывается в качестве прототипа производного объекта. Пример:

```
let objA = { a: 10 };
let objB = { b: 55 };
Object.setPrototypeOf(objB, objA);
console.log(objB.a);           // 10
console.log(objB.b);           // 55
console.log( Object.getPrototypeOf(objB) ); // { a: 10 }
```

Эту цепочку наследования мы можем еще больше расширить:

```
let objA = { a: 10 };
let objB = { b: 55 };
```

```

let objC = { c: 8 };
Object.setPrototypeOf(objB, objA);
Object.setPrototypeOf(objC, objB);
console.log(objC.a);           // 10
console.log(objC.b);           // 55
console.log(objC.c);           // 8

```

Наследование в языке JavaScript построено на прототипах. Все прототипы объектов по умолчанию ссылаются на прототип класса `Object`. Пример:

```

console.log(
  Object.getPrototypeOf(objA) === Object.prototype); // true
console.log(Object.prototype.isPrototypeOf(objA) );   // true

```

В этом примере мы воспользовались методом `isPrototypeOf(<Объект>)`, который возвращает значение `true`, если прототип входит в цепочку наследования объекта `<Объект>`, и `false` — в противном случае.

Каждый объект имеет прототип (свойство `__proto__`), который может содержать ссылку на другой объект. Этот другой объект, в свою очередь, в своем прототипе может содержать ссылку на еще один объект. И так далее. В итоге мы получаем цепочку наследования, построенную на прототипах:

```

objC
  c: 8
  __proto__:
    objB
      b: 55
      __proto__:
        objA
          a: 10
          __proto__:
            Object.prototype:
              __proto__: null
// objC => objB => objA => Object.prototype => null

```

При поиске свойства сначала просматриваются собственные свойства объекта. Если свойство найдено, то используется именно оно. В противном случае свойство ищется в прототипе объекта. Если свойство не найдено в прототипе, то просматривается вся цепочка наследования до первого совпадения. Если объекты в цепочке наследования не содержат искомого свойства, будет задействовано свойство из класса `Object`. Если свойство не найдено, мы получим значение `undefined`.

Рассмотрим наследование классов на примере (листинг 11.19).

Листинг 11.19. Наследование

```

function A(_a) {
  this.a = _a;
}

```

```
function B(_a, _b) {
    A.call(this, _a);    // Вызываем конструктор класса A
    this.b = _b;
}

function C(_a, _b, _c) {
    B.call(this, _a, _b); // Вызываем конструктор класса B
    this.c = _c;
}

B.prototype = Object.create(A.prototype);
B.prototype.constructor = B;
C.prototype = Object.create(B.prototype);
C.prototype.constructor = C;

let obj = new C(10, 55, 8);
console.log(obj.a);           // 10
console.log(obj.b);           // 55
console.log(obj.c);           // 8
console.log(obj);              // C { a: 10, b: 55, c: 8 }
console.log(obj instanceof C); // true
console.log(obj instanceof B); // true
console.log(obj instanceof A); // true
```

ОБРАТИТЕ ВНИМАНИЕ

Внутри конструкторов производных классов необходимо вызвать конструктор базового класса до первого обращения к указателю `this`. Передача указателя и параметров осуществляется с помощью метода `call()` или `apply()` (см. *разд. 10.5*).

Следующие инструкции реализуют прототипное наследование:

```
B.prototype = Object.create(A.prototype);
B.prototype.constructor = B;
C.prototype = Object.create(B.prototype);
C.prototype.constructor = C;
```

Если эти инструкции не указать, оператор `instanceof` в двух последних инструкциях из листинга 11.19 вернет значение `false`:

```
console.log(obj instanceof C); // true
console.log(obj instanceof B); // false
console.log(obj instanceof A); // false
```

11.3.5. Переопределение методов

Если имя метода в производном классе совпадает с именем метода из базового класса, то будет использоваться метод из производного класса. Чтобы вызвать метод базового класса из метода производного класса, следует воспользоваться методами `call()` или `apply()` (см. *разд. 10.5*). Метод `apply()` более универсален, т. к. ему

можно передать объект `arguments`. Рассмотрим переопределение методов на примере (листинг 11.20).

Листинг 11.20. Переопределение методов

```
function A(_a) {
    this.a = _a;
}
A.prototype.print = function() {
    console.log(`A.a = ${this.a}`);
};
function B(_a, _b) {
    A.call(this, _a);    // Вызываем конструктор класса A
    this.b = _b;
}
B.prototype = Object.create(A.prototype);
B.prototype.constructor = B;
B.prototype.print = function() {
    // Вызываем метод базового класса (если нужно, конечно)
    // A.prototype.print.apply(this, arguments);
    console.log(`B.b = ${this.b}`);
};
let objB = new B(10, 55);
objB.print(); // B.b = 55
let objA = new A(12);
objA.print(); // A.a = 12
```

Любой созданный объект автоматически наследует свойства класса `Object`. Например, при попытке вывести объект в консоли вызывается метод `toString()`, который должен возвращать значение в виде строки. Для примера выведем текущее значение:

```
let objB = new B(10, 55);
console.log(objB + '');
```

В результате в консоли получим следующий результат:

```
[object Object]
```

С помощью прототипов можно переопределить этот метод таким образом, чтобы выводилось нужное нам значение:

```
B.prototype.toString = function() {
    return `{ a: ${this.a}, b: ${this.b} }`;
}
let objB = new B(10, 55);
console.log(objB + '');
```

В результате в диалоговом окне получим следующий результат:

```
{ a: 10, b: 55 }
```

При попытке выполнить арифметическую операцию вызывается метод `valueOf()`, который должен возвращать значение в виде числа (или другого типа). Для примера переопределим метод таким образом, чтобы он возвращал значение свойства `b`:

```
B.prototype.valueOf = function() {  
    return this.b;  
}  
  
let objB = new B(10, 55);  
console.log(objB * 1);    // 55
```

11.3.6. Статические свойства и методы

Все рассмотренные варианты позволяли создавать обычные свойства и методы, вызываемые через объект. Тем не менее можно также создать свойства и методы, связанные с самим классом, а не с его объектом. Такие свойства и методы создаются непосредственно у объекта функции-конструктора и называются *статическими* (листинг 11.21).

Листинг 11.21. Статические свойства и методы

```
function Car(m) {  
    this.model = m;  
}  
  
Car.text = 'Модель ';  
Car.showInfo = function(obj) {  
    console.log(Car.text + obj.model);  
};  
  
let obj = new Car('BA3-2109');  
// Вывод значений  
console.log(Car.text); // Модель  
Car.showInfo(obj);    // Модель BA3-2109
```

Получить значение статического свойства и вызвать статический метод можно без создания объекта.

11.4. Создание класса (новый стиль)

Способы создания класса, рассмотренные в предыдущем разделе, на сегодняшний день устарели. В язык JavaScript была добавлена инструкция `class`, которую следует использовать вместо функции-конструктора. Зачем же мы тогда рассматривали старый стиль, если нужно пользоваться новым стилем? Все дело в том, что инструкция `class` всего лишь является "синтаксическим сахаром". Выглядит создание класса по-новому, а вот внутри все устроено по-старому:

```
const A = class { };  
console.log(typeof A); // function
```



```
let obj = new A();  
console.log(Object.getPrototypeOf(obj) === A.prototype); // true
```

11.4.1. Инструкция *class*

В новом стиле создание класса выполняется с помощью инструкции `class`:

```
class <Имя класса>[ extends <Базовый класс>] {  
    // <Свойства и методы класса>  
}  
  
<Переменная> = class[ <Имя класса>][ extends <Базовый класс>] {  
    // <Свойства и методы класса>  
};
```

В первом случае класс с одним именем может быть определен только один раз. Причем определение может быть расположено после инструкции создания объекта класса. Во втором случае определение класса должно быть расположено перед инструкцией создания объекта — в противном случае получите ошибку.

11.4.2. Конструктор класса

Конструктор класса внутри инструкции `class` определяется в виде метода с именем `constructor()`. Свойства объекта создаются внутри конструктора. Пример создания класса приведен в листинге 11.22.

Листинг 11.22. Инструкция `class`

```
class Car {  
    constructor(m, y) { // Конструктор класса  
        // Свойства  
        this.model = m;  
        this._year = y;  
    }  
    // Обычный метод  
    getModel() {  
        return this.model;  
    }  
    // Setter  
    set year(value) {  
        this._year = value;  
    }  
    // Getter  
    get year() {  
        return this._year;  
    }  
}
```

```
// Создание объектов
let obj1 = new Car('BA3-2109', 2007);
let obj2 = new Car('LADA Granta', 2014);
// Вывод значений
console.log( obj1._year );      // 2007
console.log( obj1.getModel() ); // BA3-2109
console.log( obj2._year );      // 2014
console.log( obj2.getModel() ); // LADA Granta
// Изменение значения с помощью setter
obj1.year = 2009;
// Получение значения с помощью getter
console.log( obj1.year );      // 2009
```

Все методы, определенные внутри класса, добавляются в прототип:

```
console.log( Object.getOwnPropertyNames(obj1) );
// [ 'model', '_year' ]
console.log( Object.getOwnPropertyNames(
    Object.getPrototypeOf(obj1) ) );
// [ 'constructor', 'getModel', 'year' ]
```

11.4.3. Наследование

Ключевое слово `extends` позволяет создать цепочку наследования. *Наследование* — это возможность создания производных классов на основе базового класса. При этом производный класс получает возможности базового класса и может добавить новую функциональность или переопределить какие-либо методы. Пример наследования классов приведен в листинге 11.23.

Листинг 11.23. Наследование

```
class A {                                // Базовый класс
  constructor(_a) {                      // Конструктор
    this.a = _a;
    console.log('Конструктор класса A');
  }
}

class B extends A {                     // Производный класс
  constructor(_a, _b) {                 // Конструктор
    super(_a);                          // Вызов конструктора базового класса
    this.b = _b;
    console.log('Конструктор класса B');
  }
}

let objB = new B(10, 55);
// Конструктор класса A
// Конструктор класса B
console.log( objB.a ); // 10
```

```
console.log( objB.b ); // 55
console.log( objB );   // B { a: 10, b: 55 }
```

В этом примере класс `B` наследует все члены класса `A`. Класс `A` называется *базовым* классом или *суперклассом*, а класс `B` — *производным* классом или *подклассом*.

ОБРАТИТЕ ВНИМАНИЕ

Внутри конструктора производного класса необходимо вызвать конструктор базового класса до первого обращения к указателю `this`. Конструктор в базовом классе автоматически не вызывается, если он был переопределен в производном классе.

Для передачи значений конструктору базового класса предусмотрен следующий синтаксис:

```
super([<Параметры вызываемого конструктора через запятую>]);
```

11.4.4. Переопределение методов

Если имя метода в производном классе совпадает с именем метода из базового класса, то будет использоваться метод из производного класса. Чтобы вызвать метод базового класса из метода производного класса, следует воспользоваться следующим синтаксисом:

```
super.<Имя метода>([<Значения>]);
```

Рассмотрим переопределение методов на примере (листинг 11.24).

Листинг 11.24. Переопределение методов

```
class A {
  constructor(_a) {
    this.a = _a;
  }
  func1(x) {
    console.log(`Метод func1() из класса A; x = ${x}`);
  }
  func2() {
    console.log('Метод func2() из класса A');
  }
}

class B extends A {
  constructor(_a) {
    super(_a); // Вызов конструктора базового класса
  }
  func1(x) {
    super.func1(x); // Вызов метода базового класса
    console.log('Метод func1() из класса B');
  }
}
```

```
func2() {
    console.log('Метод func2() из класса B');
}
}
let objB = new B(10);
objB.func1(31);
// Метод func1() из класса A; x = 31
// Метод func1() из класса B
objB.func2(); // Метод func2() из класса B
let objA = new A(5);
objA.func2(); // Метод func2() из класса A
```

11.4.5. Публичные свойства

В предыдущих примерах мы создавали свойства внутри конструктора. Однако существует возможность объявления свойств вне конструктора. Такие свойства являются *публичными* и доступны как изнутри, так и извне класса. Они инициализируются до вызова конструктора (однако внутри конструктора можно изменить их значения). Если публичному свойству не присвоено значение, оно получит значение `undefined`. Пример использования публичных свойств приведен в листинге 11.25.

Листинг 11.25. Публичные свойства

```
const name = 'd';
class A {
    a = 10;           // Публичные свойства
    b = 32;
    c;               // undefined
    [name] = 47;      // Динамическое создание имени свойства
    constructor(_a) {
        this.a = _a;  // Замена значения публичного свойства
    }
    getB() {
        return this.b;
    }
}
let objA = new A(6);
console.log( objA.a );    // 6
console.log( objA.b );    // 32
console.log( objA.getB() ); // 32
console.log( objA.c );    // undefined
console.log( objA[name] ); // 47
console.log( objA );      // A { a: 6, b: 32, c: undefined, d: 47 }
```

11.4.6. Приватные свойства и методы

Приватное свойство доступно только изнутри класса (попытка получить к нему доступ извне класса приведет к исключению). Приватное свойство должно быть объявлено вне конструктора, а перед его именем должен быть поставлен символ # (листинг 11.26).

Листинг 11.26. Приватные свойства

```
class A {
  #a = 10;           // Приватные свойства
  #b = 32;
  #c;               // undefined
  constructor(_a) {
    this.#a = _a;    // Замена значения приватного свойства
    // this.#d = 0;  // Так нельзя!
    // SyntaxError: Private field '#d' must be declared
    // in an enclosing class
  }
  print() {
    console.log(`#a = ${this.#a}`); // #a = 6
    console.log(`#b = ${this.#b}`); // #b = 32
    console.log(`#c = ${this.#c}`); // #c = undefined
  }
}

let objA = new A(6);
// console.log( objA.#a ); // Так нельзя!
// SyntaxError: Private field '#a' must be declared
// in an enclosing class
objA.print();
console.log( objA );           // A {}
console.log(
  Object.getOwnPropertyDescriptors( objA ) ); // {}
```

НА ЗАМЕТКУ

Имя приватного свойства нельзя создать динамически с помощью квадратных скобок, также нельзя обратиться к приватным свойствам с помощью квадратных скобок.

Также можно создать приватный метод, предварив его имя символом #. Создадим приватный метод и дополнительно реализуем публичный доступ к приватному свойству с помощью setter и getter (листинг 11.27).

Листинг 11.27. Приватные свойства и методы

```
class A {
  #x = 10;           // Приватное свойство
  set x(value) {     // Setter
```

```
        this.#x = value;
    }
    get x() {                                // Getter
        return this.#x;
    }
    #privateFunc (msg) {                    // Приватный метод
        console.log(`${msg}${this.#x}`);
    };
    print() {
        this.#privateFunc('#x = ');
    }
}
let objA = new A();
console.log(objA.x); // 10 (setter)
objA.x = 2;          // (getter)
objA.print();        // #x = 2
// Так нельзя
// objA.#privateFunc('test ');
// SyntaxError: Private field '#privateFunc'
// must be declared in an enclosing class
```

ОБРАТИТЕ ВНИМАНИЕ

На данный момент приватные свойства и методы не поддерживаются Mozilla Firefox (однако поддерживаются другими веб-браузерами, в частности Google Chrome).

11.4.7. Статические свойства и методы

В классе можно создать статические свойства и методы, указав перед их именами ключевое слово `static`.

ОБРАТИТЕ ВНИМАНИЕ

Внутри статических методов нет доступа к обычным свойствам и методам.

В листинге 11.28 мы передаем объект в качестве параметра статического метода. Указатель `this` внутри статического метода ссылается на класс, а не на объект. Через него мы можем обратиться к статическим членам класса. Статическое свойство можно добавить вне блока класса, указав перед его именем имя класса и оператор `точка`.

Листинг 11.28. Статические свойства и методы

```
class Car {
    static msg = 'Модель: ';                // Статическое свойство
    constructor(m) {
        this.model = m;
        console.log(this);
        // Car { model: 'ВАЗ-2109' }
    }
}
```

```

static showInfo(obj) {          // Статический метод
    console.log(`${this.msg} ${obj.model}`);
    console.log(this);
    // [class Car] { msg: 'Модель:', test: 'test' }
}
info() {
    console.log(`${Car.msg} ${this.model}`);
    console.log(`${this.constructor.msg} ${this.model}`);
}
}
Car.test = 'test';              // Статическое свойство
let obj = new Car('BA3-2109');
Car.showInfo(obj);              // Модель: BA3-2109
console.log(Car.msg);           // Модель:
console.log(Car.test);          // test
obj.info();
// Модель: BA3-2109
// Модель: BA3-2109

```

Внутри обычного метода обращение к статическим членам осуществляется с помощью имени класса или свойства `constructor`:

```

console.log(`${Car.msg} ${this.model}`);
console.log(`${this.constructor.msg} ${this.model}`);

```

Статические свойства можно сделать приватными (листинг 11.29), добавив перед именем свойства символ `#`. Обращаться к приватным свойствам допускается только внутри методов класса.

Листинг 11.29. Приватные статические свойства и методы

```

class Car {
    static #msg = 'Модель:';      // Приватное статическое свойство
    static #test = function() {   // Приватный статический метод
        console.log('test()');
    };
    constructor(m) {
        this.model = m;
    }
    showInfo() {
        console.log(`${Car.#msg} ${this.model}`);
        Car.#test();
    }
}
let obj = new Car('BA3-2109');
obj.showInfo();
// Модель: BA3-2109
// test()

```

```
// console.log(Car.#msg);  
// SyntaxError: Private field '#msg' must be declared  
// in an enclosing class  
// Car.#test();  
// SyntaxError: Private field '#test' must be declared  
// in an enclosing class
```

11.5. Оператор *instanceof*

С помощью оператора `instanceof` можно проверить принадлежность объекта какому-либо классу:

```
let k = new Number(8);  
if ( (typeof k === 'object') && (k instanceof Number) )  
    console.log('Объект k принадлежит классу Number');  
else console.log('Нет');
```

Оператор `instanceof` также проверяет цепочку наследования (листинг 11.30).

Листинг 11.30. Оператор `instanceof`

```
class A {  
    constructor(_a) {  
        this.a = _a;  
    }  
}  
  
class B extends A {  
    constructor(_a, _b) {  
        super(_a);  
        this.b = _b;  
    }  
}  
  
let objB = new B(10, 55);  
console.log(objB instanceof B); // true  
console.log(objB instanceof A); // true  
let objA = new A(6);  
console.log(objA instanceof A); // true  
console.log(objA instanceof B); // false  
console.log( A.prototype.isPrototypeOf(objB) ); // true  
console.log( B.prototype.isPrototypeOf(objB) ); // true  
console.log( A.prototype.isPrototypeOf(objA) ); // true  
let objC = {};  
console.log( A.prototype.isPrototypeOf(objC) ); // false
```

В этом примере мы воспользовались методом `isPrototypeOf(<Объект>)`, который возвращает значение `true`, если прототип входит в цепочку наследования объекта `<Объект>`, и `false` — в противном случае.

11.6. Массивоподобные объекты (псевдомассивы)

Массивоподобные объекты (псевдомассивы) поддерживают функциональность массивов. Как и массивы, они хранят набор значений (элементов) и позволяют обращаться к ним, указав индекс внутри квадратных скобок. В отличие от массивов, такие объекты не являются наследниками класса `Array` и не содержат его методов.

Примером массивоподобного объекта является объект `arguments`, позволяющий внутри функции получить доступ ко всем переданным параметрам. Использование объекта `arguments` показано в листинге 11.31.

Листинг 11.31. Использование объекта `arguments`

```
function test() {  
    console.log(arguments);  
    console.log(arguments.length);  
    console.log(arguments[0]);  
}  
test(55);  
// [Arguments] { '0': 55 }  
// 1  
// 55  
test(80, 66, 17);  
// [Arguments] { '0': 80, '1': 66, '2': 17 }  
// 3  
// 80
```

Создадим массивоподобный объект:

```
let obj = {  
    0: 20,  
    1: 30,  
    2: 40,  
    length: 3  
};  
console.log( obj[0] ); // 20  
console.log( obj[1] ); // 30  
console.log( obj[2] ); // 40  
for (let i = 0; i < obj.length; i++) {  
    console.log( obj[i] );  
}
```

Для преобразования массивоподобного объекта в полноценный массив можно воспользоваться статическим методом `from()` из класса `Array`:

```
let arr = Array.from(obj);  
console.log(arr);      // [ 20, 30, 40 ]
```

Если объект поддерживает итерации, можно воспользоваться оператором `SPREAD`. Наш объект не поддерживает итерации, поэтому получим сообщение об ошибке, а вот объект `arguments` итерации поддерживает. Пример:

```
function test() {
  let arr1 = Array.from(arguments);
  console.log(arr1);      // [ 10, 20, 30 ]
  let arr2 = [...arguments];
  console.log(arr2);      // [ 10, 20, 30 ]
}
test(10, 20, 30);
```

Что делать, если нужно использовать методы класса `Array`, а преобразовывать объект в массив не хочется? Для этого можно воспользоваться методом `call()`. Пример:

```
let obj = {
  0: 20,
  1: 30,
  2: 40,
  length: 3
};
console.log( Array.prototype.indexOf.call(obj, 40) ); // 2
console.log( [].indexOf.call(obj, 30) );              // 1
```

Совместное использование методов `slice()` и `call()` позволяет преобразовать массивоподобный объект в настоящий массив:

```
let arr1 = Array.prototype.slice.call(obj);
console.log(arr1);      // [ 20, 30, 40 ]
let arr2 = [].slice.call(obj);
console.log(arr2);      // [ 20, 30, 40 ]
```

11.7. Итераторы

Многие встроенные классы поддерживают перебор с помощью *итераторов*. Благодаря этому, например, мы можем перебрать массив с помощью цикла `for...of`. Пример:

```
let arr = [1, 2, 3];
for (let value of arr) {
  process.stdout.write(`${value} `);
} // 1 2 3
```

или преобразовать строку в массив букв с помощью оператора `SPREAD`:

```
let str = 'string';
let arr = [...str];
console.log(arr);    // [ 's', 't', 'r', 'i', 'n', 'g' ]
```

Объект, поддерживающий итерации, возвращают также функции-генераторы:

```
function* gen() {
  yield 10;
  yield 20;
  yield 30;
}

let arr = [...gen()];
console.log(arr);    // [ 10, 20, 30 ]
let it = gen();
console.log(it);     // Object [Generator] {}
```

Объект-итератор содержит метод `next()`, при каждом вызове возвращающий объект с двумя свойствами: `value` (хранит текущее значение) и `done` (`true`, если больше нет значений, и `false` — в противном случае):

```
let map = new Map();
map.set('a', 1).set('b', 2);
let it = map.keys();
console.log(it);           // [Map Iterator] { 'a', 'b' }
console.log(it.next());    // { value: 'a', done: false }
console.log(it.next());    // { value: 'b', done: false }
console.log(it.next());    // { value: undefined, done: true }
it = map.keys();
console.log(it.next().value); // a
console.log(it.next().value); // b
```

Создать итератор также можно с помощью символа `Symbol.iterator`:

```
let map = new Map();
map.set('a', 1).set('b', 2);
let it = map[Symbol.iterator]();
console.log(it);           // [Map Entries] { [ 'a', 1 ], [ 'b', 2 ] }
console.log(it.next());    // { value: [ 'a', 1 ], done: false }
console.log(it.next());    // { value: [ 'b', 2 ], done: false }
console.log(it.next());    // { value: undefined, done: true }
```

Свойство `Symbol.iterator` должно содержать ссылку на функцию, которая возвращает объект, поддерживающий итерации:

```
let obj = {};
obj[Symbol.iterator] = function*() {
  yield 10;
  yield 20;
};
console.log([...obj]); // [ 10, 20 ]
```

Цикл `for...of` сначала как раз и вызывает метод `Symbol.iterator`, а затем работает с возвращаемым им объектом, содержащим метод `next()`. В предыдущем разделе мы создали массивоподобный объект, который не поддерживает итерации. Давайте сделаем этот объект итерируемым (листинг 11.32).

Листинг 11.32. Итерируемые массивоподобные объекты

```
let obj = {
  0: 20,
  1: 30,
  2: 40,
  length: 3,
  [Symbol.iterator]: function() {
    return {
      _index: 0,
      _arr: this,
      next() {
        if (this._index < this._arr.length) {
          return {
            done: false,
            value: this._arr[this._index++]
          };
        }
        else {
          return { done: true, value: undefined };
        }
      },
      toString() {
        return 'MyIterator {}';
      }
    };
  }
};

console.log( [...obj] ); // [ 20, 30, 40 ]
let it = obj[Symbol.iterator]();
console.log( it.toString() ); // MyIterator {}
console.log( it.next() );    // { done: false, value: 20 }
console.log( it.next() );    // { done: false, value: 30 }
console.log( it.next() );    // { done: false, value: 40 }
console.log( it.next() );    // { done: true, value: undefined }
for (let value of obj) {
  console.log(value);
}
```

11.8. Методы-генераторы

В классах можно определять *методы-генераторы*, подобные функциям-генераторам (см. *разд. 10.11*). В начале имени метода-генератора должен стоять символ звездочки (*).

Напишем класс `Counter` с методом `numbers(<Начальное значение>, <Конечное значение>)`, который выдает последовательность четных целых чисел, расположенных между заданными значениями (листинг 11.33).

Листинг 11.33. Метод-генератор

```

class Counter {
  *evens(begin, end) {
    for (let i = begin; i <= end; i++) {
      if (i % 2 == 0) {
        yield i;
      }
    }
  }
}

const cnt = new Counter();
for (let n of cnt.evens(1, 5)) {
  console.log(n);
}
// 2
// 4

```

Можно объявлять приватные, статические и приватные статические методы-генераторы. В именах приватных и приватных статических методов-генераторов символ `*` должен стоять перед символом `#`. Пример:

```

class SomeClass {
  // Приватный метод-генератор
  *#method1() { . . . }
  // Публичный статический метод-генератор
  static *staticMethod2() { . . . }
  // Приватный статический метод-генератор
  static *#staticMethod3() { . . . }
}

```

ОБРАТИТЕ ВНИМАНИЕ

Не забываем, что на данный момент приватные свойства и методы не поддерживаются Mozilla Firefox (однако поддерживаются другими веб-браузерами, в частности Google Chrome).

11.9. Пространства имен

Предположим, программист написал функцию `inArray()`. Через некоторое время потребовалось подключить библиотеку стороннего разработчика, в которой все функции объявлены в глобальной области видимости. Если в этой библиотеке объявлена функция с именем `inArray()`, возникнет конфликт имен. Никакого сообщения об ошибке в данном случае выведено не будет, просто функция, объявленная последней, переопределит ранее объявленную функцию. В итоге будет получен результат, который не планировался, или программа завершится с критической ошибкой.

Чтобы избежать подобной ситуации, следует строго придерживаться концепции *пространств имен*, согласно которой любой модуль может импортировать в глобальную область видимости только один идентификатор.

В языке JavaScript в качестве пространств имен используются объекты. Объект, представляющий пространство имен, помещается в глобальную область видимости, а остальные идентификаторы доступны через свойства этого объекта. Пример:

```
let myModule = {}; // Объявление пространства имен
myModule.test = function() {
    console.log('Это функция test');
}
myModule.inArray = function() {
    console.log('Это функция inArray');
}
myModule.test();
myModule.inArray();
```

В этом примере функция `inArray()` расположена внутри пространства имен `myModule`, поэтому конфликт имен сводится к минимуму. Однако может возникнуть ситуация, когда пространства имен называются одинаково. В этом случае решением является создание вложенных объектов. Очень часто имя пространства имен совпадает с названием сайта разработчика. В качестве основного объекта используется название зоны, а вложенный объект носит название домена. Например, для сайта **<http://example.com/>** создание пространства имен будет выглядеть так:

```
var com; // Объявляем, иначе будет ошибка при проверке
if (!com) com = {}; // Объявление пространства имен
else if (typeof com !== 'object')
    throw new Error('Идентификатор com не является объектом');
if (com.example)
    throw new Error('Пространство имен уже занято');
com.example = { // Объявление вложенного пространства имен
    test: function() {
        console.log('Это функция test');
    },
    inArray: function() {
        console.log('Это функция inArray');
    }
};
com.example.test();
com.example.inArray();
```

Таким образом, если домен принадлежит вам, никакого конфликта не будет, но пользоваться таким длинным именем не очень удобно. Учитывая, что присваивание объектов осуществляется по ссылке, а не по значению, данная проблема решается просто. В программе определяется короткий идентификатор, и в нем сохраняется ссылка на объект. Пример:

```
let $ = com.example;
$.test();
$.isArray();
```

Кроме того, можно использовать анонимную функцию, в параметре которой указывается короткий идентификатор, а при вызове функции передается ссылка на объект, являющийся пространством имен:

```
(function($) {
    $.test();
    $.isArray();
})(com.example);
```

В этом примере идентификатор `$` будет доступен только внутри анонимной функции, а, т. к. функция не имеет имени, в глобальной области видимости никакой идентификатор не сохраняется.

11.10. Класс *Proxy*

Класс `Proxy` позволяет реализовать "обертку" над объектом, перехватывающую различные операции над ним. Например, можно полностью контролировать обращения к свойствам объекта, реализуя один из принципов ООП, который называется *инкапсуляцией* (сокрытие данных внутри класса).

11.10.1. Создание проху-объекта

Создать объект класса `Proxy` позволяет следующий конструктор:

```
<Проху-объект> = new Proxy(<Исходный объект>, <Объект с ловушками>)
```

В первом параметре указывается объект, над которым нужно реализовать "обертку". Это может быть обычный объект, массив, функция или другой проху-объект. Во втором параметре указывается объект с методами, имеющими специальные имена. Эти методы будут вызваны при попытке доступа к исходному объекту через проху-объект. В итоге мы получаем ссылку на проху-объект.

Если во втором параметре указать пустой объект, то все операции будут перенаправлены исходному объекту:

```
let obj = { a: 2 };
let proxy = new Proxy(obj, {});
console.log( proxy.a ); // 2
proxy.a = 5;
proxy.b = 10;
console.log( proxy );    // { a: 5, b: 10 }
delete proxy.a;
for (let prop in proxy) {
    console.log(prop);    // b
}
```

В этом примере мы можем обратиться к исходному объекту напрямую, минуя проху-объект:

```
console.log( obj );      // { b: 10 }
obj.c = 3;
console.log( proxy );    // { b: 10, c: 3 }
```

Следует учитывать, что при обращении к исходному объекту напрямую методы из объекта с ловушками вызваны не будут. Поэтому старайтесь не оставлять ссылок на исходные объекты. Пример:

```
let proxy = new Proxy({ a: 2 }, {});
proxy.a = 5;
proxy.b = 10;
console.log( proxy );    // { a: 5, b: 10 }
```

Вместо простого объекта в первом параметре можно указать массив:

```
let proxy = new Proxy([1, 2, 3], {});
proxy[0] = 5;
delete proxy[2];
console.log( proxy );    // [ 5, 2, <1 empty item> ]
```

или обернуть ссылку на функцию:

```
let proxy = new Proxy( function() {
  console.log('test');
}, {});
proxy();    // test
```

Пример указания другого проху-объекта:

```
let obj = { a: 2 };
let proxy = new Proxy(obj, {});
let proxy2 = new Proxy(proxy, {});
proxy2.a = 10;
console.log(obj.a); // 10
```

При попытке создать проху-объект над объектами некоторых встроенных классов (например, Map или Set), а также при использовании внутри объекта приватных свойств при попытке доступа к свойствам может возникнуть исключение:

```
let proxy = new Proxy(new Set(), {});
proxy.add('a');
// TypeError: Method Set.prototype.add called on incompatible
// receiver [object Object]
```

Эти объекты используют внутренние слоты, и при создании проху-объекта теряется контекст, с помощью которого можно получить доступ к свойствам. Чтобы контекст не терялся, нужно передать его явным образом с помощью метода bind(). Использование этого метода иллюстрирует листинг 11.34.

Листинг 11.34. Использование метода bind() при создании проху-объекта

```

let map = new Map();
let proxy = new Proxy(map, {
  get(target, prop, receiver) {
    console.log('get', prop);
    let result = Reflect.get(target, prop, receiver);
    if (typeof result == 'function')
      return result.bind(target);
    return result;
  }
});
proxy.set('a', 2);
console.log( proxy.get('a') ); // 2

```

11.10.2. Ограничение доступа к свойствам объекта

Чтобы контролировать значения, присваиваемые свойствам, следует внутри объекта с ловушками определить метод `set()`. Формат метода:

```

set(target, property, value, receiver) {
  return <true или false>;
}

```

Через первый параметр доступна ссылка на исходный объект, через второй — имя свойства (в виде строки, числа или `symbol`), через третий — новое значение свойства, а через четвертый — ссылка на проху-объект или объект, унаследованный от проху-объекта. Метод `set()` должен вернуть значение `true`, если значение свойства успешно изменено, и `false` — в противном случае. В обычном режиме возврат значения `false` игнорируется. При использовании строгого режима (указана инструкция `'use strict';`) возврат значения `false` приведет к генерации исключения `TypeError`. Пример ограничения изменения значений у свойств объекта приведен в листинге 11.35.

Листинг 11.35. Метод set()

```

// 'use strict';
let obj = { a: 2, _n: 3 };
let proxy = new Proxy(obj, {
  set(target, prop, value, receiver) {
    console.log('Вызван метод set()');
    // console.log(target === obj); // true
    // console.log(receiver === proxy); // true
    if (prop in target) {
      if (prop === 'a' && typeof value === 'number') {
        if (value >= 0 && value <= 10) {
          target[prop] = value;
          return true;
        }
      }
    }
  }
}

```

```

        else {
            console.error('value >= 0 && value <= 10');
            return false;
        }
    }
    else {
        console.error('Нельзя изменить значение свойства');
        return false;
    }
}
else {
    // Добавление нового свойства
    target[prop] = value;
    return true;
}
}
});
proxy.a = 5;      // Вызван метод set()
console.log(obj); // { a: 5, _n: 3 }
proxy.a = 20;     // Вызван метод set()
                  // value >= 0 && value <= 10
proxy._n = 20;    // Вызван метод set()
                  // Нельзя изменить значение свойства
proxy.b = 40;     // Вызван метод set()
console.log(obj); // { a: 5, _n: 3, b: 40 }
// При доступе через исходный объект метод set() не вызывается!
obj._n = 55;
console.log(obj); // { a: 5, _n: 55, b: 40 }

```

Чтобы перехватывать попытки получить значения свойств, следует внутри объекта с ловушками определить метод `get()`. Формат метода:

```

get(target, property, receiver) {
    return <Значение>;
}

```

Через первый параметр доступна ссылка на исходный объект, через второй — имя свойства (в виде строки, числа или `symbol`), а через третий — ссылка на прокси-объект или объект, унаследованный от прокси-объекта. Метод `get()` должен вернуть значение свойства, `undefined` или какое-либо значение по умолчанию. Пример ограничения возврата значений из свойств объекта приведен в листинге 11.36.

Листинг 11.36. Метод `get()`

```

let obj = { a: 2, _n: 3 };
let proxy = new Proxy(obj, {
    get(target, prop, receiver) {

```

```

    // console.log('Вызван метод get()');
    // console.log(target === obj);      // true
    // console.log(receiver === proxy); // true
    if (prop in target) {
        if (prop === 'a') {
            return target[prop];
        }
        else {
            return undefined;
        }
    }
    else {
        return 'Значение по умолчанию';
    }
}
});
console.log(proxy.a); // 2
console.log(proxy._n); // undefined
console.log(proxy.c); // Значение по умолчанию
// При доступе через исходный объект метод get() не вызывается!
console.log(obj._n); // 3
console.log(obj.c); // undefined

```

В листингах 11.35 и 11.36 параметр `receiver` ссылается на `проху`-объект. Но если от `проху`-объекта наследуется другой объект, то этот параметр будет ссылаться на следующий объект (такой случай показан в листинге 11.37).

Листинг 11.37. Случай с наследованием от `проху`-объекта

```

let obj = { a: 2, _n: 3 };
let proxy = new Proxy(obj, {
    get(target, prop, receiver) {
        console.log(receiver === proxy); // false
        console.log(receiver === obj2); // true
        return target[prop];
    },
    set(target, prop, value, receiver) {
        console.log(receiver === proxy); // false
        console.log(receiver === obj2); // true
        target[prop] = value;
        return true;
    }
});
let obj2 = { b: 47 };
Object.setPrototypeOf(obj2, proxy);
obj2.a = 5;
console.log(obj2.a); // 5

```

Для выполнения обычного доступа к свойствам можно воспользоваться статическими методами `set()` и `get()` из класса `Reflect`. Форматы методов:

```
<true или false> = set(target, property, value[, receiver])  
<Значение> = get(target, property[, receiver])
```

Все параметры совпадают с одноименными параметрами методов `set()` и `get()` внутри объекта с ловушками. Возвращаемые ими значения можно вернуть внутри этих методов. Применение методов `get()` и `set()` из класса `Reflect` показано в листинге 11.38.

Листинг 11.38. Применение методов `get()` и `set()` из класса `Reflect`

```
let obj = { a: 2 };  
let proxy = new Proxy(obj, {  
  get(target, prop, receiver) {  
    // return target[prop];  
    return Reflect.get(target, prop, receiver);  
  },  
  set(target, prop, value, receiver) {  
    // target[prop] = value;  
    // return true;  
    return Reflect.set(target, prop, value, receiver);  
  }  
});  
proxy.a = 5;  
console.log(proxy.a); // 5
```

11.10.3. Ограничение проверки существования свойства

При использовании оператора `in` внутри объекта с ловушками вызывается метод `has()`. Формат метода:

```
has(target, name) {  
  return <true или false>;  
}
```

Через первый параметр доступна ссылка на исходный объект, а через второй — имя проверяемого свойства (в виде строки, числа или `symbol`). Метод `has()` должен вернуть значение `true`, если свойство существует, и `false` — в противном случае.

Вернем значение `false` для всех свойств, имена которых начинаются с подчеркивания (листинг 11.39).

Листинг 11.39. Метод `has()`

```
let obj = { a: 2, _n: 3 };  
let proxy = new Proxy(obj, {
```

```

    has(target, name) {
        if ( name.startsWith('_') ) return false;
        return name in target;
    }
});
console.log('a' in proxy); // true
console.log('_n' in proxy); // false
// При доступе через исходный объект метод has() не вызывается!
console.log('_n' in obj); // true

```

11.10.4. Ограничение получения списка свойств и их атрибутов

При попытке получения списка имен свойств объекта вызывается метод `ownKeys()` из объекта с ловушками. Формат метода:

```

ownKeys(target) {
    return <Массив>;
}

```

Метод `ownKeys()` вызывается при использовании цикла `for...in`, статических методов `keys()`, `values()`, `entries()`, `getOwnPropertyNames()` и `getOwnPropertySymbols()` из класса `Object`. Через параметр доступна ссылка на исходный объект. Метод должен вернуть массив с именами свойств.

Вернем массив с именами всех собственных перечислимых свойств, кроме свойств, имена которых начинаются с подчеркивания (листинг 11.40).

Листинг 11.40. Метод `ownKeys()`

```

let obj = { a: 2, _n: 3 };
let proxy = new Proxy(obj, {
    ownKeys(target) {
        return Object.keys(target)
            .filter( name => !name.startsWith('_') );
    }
});
console.log( Object.keys(proxy) ); // [ 'a' ]
console.log( Object.values(proxy) ); // [ 2 ]
console.log( Object.entries(proxy) ); // [ [ 'a', 2 ] ]
console.log( Object.getPrototypeOf(proxy) ); // [ 'a' ]
console.log( Object.getPrototypeOf(proxy) ); // []
for (const name in proxy) {
    console.log(name);
} // a
// При доступе через исходный объект метод ownKeys() не вызывается!
console.log( Object.keys(obj) ); // [ 'a', '_n' ]

```

ОБРАТИТЕ ВНИМАНИЕ

Если в возвращаемый массив добавить имя несуществующего свойства в виде строки, оно будет получено только при использовании метода `getOwnPropertyNames()`.

Остальные методы и цикл `for...in` дополнительно проверяют, чтобы свойство было перечислимым (атрибут `enumerable` имел значение `true`). Проверка выполняется с помощью метода `getOwnPropertyDescriptor()`. Если внутри объекта с ловушками определить одноименный метод, то проверку можно будет перехватить. Формат метода:

```
getOwnPropertyDescriptor(target, name) {  
    return <Объект с атрибутами или undefined>  
}
```

Через первый параметр доступна ссылка на исходный объект, а через второй — имя свойства (в виде строки, числа или `symbol`). Метод должен вернуть объект (дескриптор) со всеми атрибутами собственного свойства исходного объекта. Если свойство не найдено, метод должен вернуть значение `undefined`. Пример реализации метода `getOwnPropertyDescriptor()` приведен в листинге 11.41.

Листинг 11.41. Метод `getOwnPropertyDescriptor()`

```
let obj = { a: 2, _n: 3 };  
let proxy = new Proxy(obj, {  
    ownKeys(target) {  
        return ['a', 'b'];  
    },  
    getOwnPropertyDescriptor(target, name) {  
        if ( name.startsWith('_') ) return undefined;  
        if (name === 'b') {  
            return {  
                value: 10, writable: false, enumerable: true,  
                configurable: true  
            };  
        }  
        return Reflect.getOwnPropertyDescriptor(target, name);  
    },  
    get(target, prop, receiver) {  
        if ( prop.startsWith('_') ) return undefined;  
        if (prop in target) return Reflect.get(target, prop, receiver);  
        if (prop === 'b') return 10;  
        return undefined;  
    }  
});  
  
console.log( Object.keys(proxy) );    // [ 'a', 'b' ]  
console.log( Object.values(proxy) );  // [ 2, 10 ]  
console.log( Object.entries(proxy) ); // [ [ 'a', 2 ], [ 'b', 10 ] ]  
console.log( Object.getPrototypeOf(proxy) ); // [ 'a', 'b' ]
```

```

for (const name in proxy) {
    console.log(name);
}
// a
// b
console.log( proxy._n );    // undefined
console.log( Object.getOwnPropertyDescriptor(proxy, 'a') );
// { value: 2, writable: true, enumerable: true, configurable: true }
console.log( Object.getOwnPropertyDescriptor(proxy, '_n') );
// undefined
console.log( Object.getOwnPropertyDescriptor(proxy, 'b') );
// { value: 10, writable: false, enumerable: true, configurable: true }

```

В этом примере мы воспользовались методом `getOwnPropertyDescriptor(target, name)` из класса `Reflect`, который аналогичен одноименному методу из класса `Object`.

11.10.5. Ограничение добавления свойства и изменения атрибутов

При добавлении свойства с помощью точечной нотации или при использовании методов `defineProperty()` и `defineProperties()` внутри объекта с ловушками вызывается метод `defineProperty()`. Формат метода:

```

defineProperty(target, property, attributes) {
    return <true или false>;
}

```

Через первый параметр доступна ссылка на исходный объект, через второй — имя свойства (в виде строки, числа или `symbol`), а через третий — объект с атрибутами. При удачно выполненной операции метод должен вернуть значение `true`. Если метод вернет `false` или `undefined`, будет сгенерировано исключение `TypeError`.

Пример реализации метода `defineProperty()` приведен в листинге 11.42.

Листинг 11.42. Метод `defineProperty()`

```

let obj = { a: 2, _n: 3 };
let proxy = new Proxy(obj, {
    defineProperty(target, prop, attributes) {
        console.log('Вызван метод defineProperty()');
        if ( prop.startsWith('_') ) return false;
        return Reflect.defineProperty(target, prop, attributes);
    }
});
Object.defineProperty(proxy, 'b', {
    value: 22,
    writable: true,

```

```

    configurable: true,
    enumerable: true
  });
// Вызван метод defineProperty()
console.log( proxy.b ); // 22
// Object.defineProperty(proxy, '_n', { value: 34 }); // TypeError
proxy.c = 20;
// Вызван метод defineProperty()
console.log( proxy.c ); // 20
Object.defineProperties(proxy, { 'a': { enumerable: false } });
// Вызван метод defineProperty()

```

В этом примере мы воспользовались методом `defineProperty(target, property, attributes)` из класса `Reflect`, который аналогичен одноименному методу из класса `Object`. Метод возвращает значение `true`, если операция выполнена успешно, и `false` — в противном случае.

11.10.6. Ограничение удаления свойства

При использовании оператора `delete` внутри объекта с ловушками вызывается метод `deleteProperty()`. Формат метода:

```

deleteProperty(target, property) {
  return <true или false>;
}

```

Через первый параметр доступна ссылка на исходный объект, а через второй — имя свойства (в виде строки, числа или `symbol`). При удачно выполненной операции метод должен вернуть значение `true`, а в противном случае — `false`. В обычном режиме возврат значения `false` игнорируется. При использовании строгого режима (включается указанием в начале файла инструкции `'use strict';`) возврат значения `false` приведет к генерации исключения `TypeError`.

Пример реализации метода `deleteProperty()` приведен в листинге 11.43.

Листинг 11.43. Метод `deleteProperty()`

```

// 'use strict';
let obj = { a: 2, _n: 3 };
let proxy = new Proxy(obj, {
  deleteProperty(target, prop) {
    if ( prop.startsWith('_') ) return false;
    return Reflect.deleteProperty(target, prop);
  }
});
console.log( delete proxy.a ); // true
console.log( delete proxy._n ); // false
console.log( obj );           // { _n: 3 }

```



```
// При доступе через исходный объект,
// метод deleteProperty() не вызывается!
console.log( delete obj._n ); // true
console.log( obj );           // {}
```

В этом примере мы воспользовались методом `deleteProperty(target, property)` из класса `Reflect`, который является аналогом оператора `delete` в нестрогом режиме. Метод возвращает значение `true`, если операция выполнена успешно, и `false` — в противном случае.

11.10.7. Ограничение нерасширяемости объекта

При вызове статического метода `preventExtensions()` из класса `Object` (запрещает расширение объекта) внутри объекта с ловушками вызывается метод `preventExtensions()`. Формат метода:

```
preventExtensions(target) {
    return <true или false>;
}
```

Через первый параметр доступна ссылка на исходный объект. При удачно выполненной операции метод должен вернуть значение `true`, а в противном случае — `false`. Если метод вернет `false`, то будет сгенерировано исключение.

При вызове статического метода `isExtensible()` из класса `Object` (возвращает `true`, если объект является расширяемым, и `false` — в противном случае) внутри объекта с ловушками вызывается метод `isExtensible()`. Формат метода:

```
isExtensible(target) {
    return <true или false>;
}
```

Через первый параметр доступна ссылка на исходный объект. Метод должен вернуть значение `true`, если объект является расширяемым, и `false` — в противном случае.

Пример реализации методов `preventExtensions()` и `isExtensible()` приведен в листинге 11.44.

Листинг 11.44. Методы `preventExtensions()` и `isExtensible()`

```
let obj = { a: 2, _n: 3 };
let proxy = new Proxy(obj, {
    preventExtensions(target) {
        console.log('preventExtensions');
        return Reflect.preventExtensions(target);
    },
    isExtensible(target) {
        console.log('isExtensible');
```

```
        return Reflect.isExtensible(target);
    }
});
console.log( Object.isExtensible(proxy) ); // true
Object.preventExtensions(proxy);
console.log( Object.isExtensible(proxy) ); // false
```

В этом примере мы воспользовались статическими методами `preventExtensions(target)` и `isExtensible(target)` из класса `Reflect`, которые аналогичны одноименным методам из класса `Object` и возвращают `true`, если операция выполнена успешно или объект расширяем, и `false` — в противном случае.

11.10.8. Ограничение доступа к прототипу

При вызове статического метода `setPrototypeOf()` из класса `Object` внутри объекта с ловушками вызывается метод `setPrototypeOf()`. Формат метода:

```
setPrototypeOf(target, proto) {
    return <true или false>;
}
```

Через первый параметр доступна ссылка на исходный объект, а через второй — прототип или значение `null`. При удачно выполненной операции метод должен вернуть значение `true`, а в противном случае — `false`. Если метод вернет `false`, будет сгенерировано исключение.

При вызове статического метода `getPrototypeOf()` из класса `Object` внутри объекта с ловушками вызывается метод `getPrototypeOf()`. Формат метода:

```
getPrototypeOf(target) {
    return <Объект>;
}
```

Через первый параметр доступна ссылка на исходный объект. Метод должен вернуть ссылку на прототип объекта или значение `null`.

НА ЗАМЕТКУ

Эти методы также вызываются при доступе через свойство `__proto__`.

Пример реализации методов `setPrototypeOf()` и `getPrototypeOf()` приведен в листинге 11.45.

Листинг 11.45. Методы `setPrototypeOf()` и `getPrototypeOf()`

```
let obj = { a: 2, _n: 3 };
let proxy = new Proxy(obj, {
    setPrototypeOf(target, proto) {
        console.log('setPrototypeOf');
        return Reflect.setPrototypeOf(target, proto);
    },
```

```

getPrototypeOf(target) {
    console.log('getPrototypeOf');
    return Reflect.getPrototypeOf(target);
}
});
Object.setPrototypeOf(proxy, { b: 10 });
console.log( Object.getPrototypeOf(proxy) ); // { b: 10 }

proxy.__proto__ = { c: 64 };
console.log( proxy.__proto__ );           // { c: 64 }

```

В этом примере мы воспользовались статическими методами `setPrototypeOf(target, proto)` и `getPrototypeOf(target)` из класса `Reflect`, аналогичными одноименным методам из класса `Object`. Метод `setPrototypeOf()` возвращает значение `true`, если операция выполнена успешно, и `false` — в противном случае. Метод `getPrototypeOf()` возвращает ссылку на прототип объекта или значение `null`.

11.10.9. Перехват создания объекта

При попытке создания объекта с помощью оператора `new` внутри объекта с ловушками вызывается метод `construct()`. Формат метода:

```

construct(target, args) {
    return <Объект>;
}

```

Через первый параметр доступна ссылка на функцию-конструктор, а через второй — массив с параметрами, переданными конструктору. При удачно выполненной операции метод должен вернуть новый объект. Пример перехвата создания объекта приведен в листинге 11.46.

Листинг 11.46. Перехват создания объекта

```

function A(_a) {
    this.a = _a;
}

let proxy = new Proxy(A, {
    construct(target, args) {
        console.log('construct');
        return Reflect.construct(target, args);
    }
});

let obj = new proxy(30);
console.log( obj );           // A { a: 30 }

```

В этом примере мы воспользовались статическим методом `construct(target, args[, newTarget])` из класса `Reflect`, который возвращает новый объект, вызывая инструкцию `new target(...args)`.

11.10.10. Перехват вызова функции

При попытке вызова функции внутри объекта с ловушками вызывается метод `apply()`. Формат метода:

```
apply(target, thisArg, args) {  
    return <Результат вызова функции>;  
}
```

Через первый параметр доступна ссылка на исходную функцию, через второй — контекст, который будет доступен внутри функции через указатель `this`, а через третий — массив с параметрами, переданными вызываемой функцией. Метод должен вернуть результат выполнения функции. Пример перехвата вызова функции приведен в листинге 11.47.

Листинг 11.47. Перехват вызова функции

```
function sum(x, y) {  
    // console.log(this);  
    return x + y;  
}  
  
let proxy = new Proxy(sum, {  
    apply(target, thisArg, args) {  
        console.log('apply', thisArg, args);  
        return Reflect.apply(target, thisArg, args);  
    }  
});  
  
console.log( proxy(30, 10) );           // 40  
// apply undefined [ 30, 10 ]  
console.log( proxy.call({ n: 8 }, 71, 3) ); // 74  
// apply { n: 8 } [ 71, 3 ]  
console.log( proxy.apply({ a: 10 }, [ 15, 6 ]) ); // 21  
// apply { a: 10 } [ 15, 6 ]
```

В этом примере мы воспользовались статическим методом `apply(target, thisArg, args)` из класса `Reflect`, который вызывает функцию и возвращает результат ее выполнения.

11.10.11. Отключаемый проху-объект

Статический метод `revocable()` из класса `Proxy` позволяет создать отключаемый проху-объект. Формат метода:

```
<Объект> = Proxy.revocable(<Исходный объект>, <Объект с ловушками>)
```

В первом параметре указывается объект, над которым нужно реализовать обертку. Это может быть обычный объект, массив, функция или другой проху-объект. Во втором параметре указывается объект с ловушками. Метод `revocable()` возвращает ссылку на объект с двумя свойствами:

- `proxy` — содержит ссылку на проху-объект;
- `revoke` — содержит ссылку на функцию, при вызове которой проху-объект будет отключен. Любые действия с отключенным проху-объектом приведут к генерации исключения.

Пример использования отключаемого проху-объекта приведен в листинге 11.48.

Листинг 11.48. Отключаемый проху-объект

```
let { proxy, revoke } = Proxy.revocable({ a: 2, b: 3 }, {
  get(target, prop, receiver) {
    console.log('get', prop);
    return Reflect.get(target, prop, receiver);
  }
});
console.log( proxy.a ); // 2
revoke();              // Отключаем проху-объект
// console.log( proxy.a );
// TypeError: Cannot perform 'get' on a proxy that has been revoked
console.log( Proxy.revocable({}, {}) );
// { proxy: {}, revoke: [Function] }
```

11.11. Класс *Reflect*

Класс `Reflect` содержит статические методы, имена которых совпадают с именами методов внутри объекта с ловушками в классе `Proxy`. Значения, возвращаемые этими методами, можно вернуть внутри одноименных методов в объекте с ловушками, реализуя действия по умолчанию. В предыдущих разделах мы уже рассмотрели примеры такого использования и познакомились почти со всеми методами класса `Reflect`. Еще раз перечислим их:

- `apply(target, thisArg, args)` — вызывает функцию `target` и возвращает результат ее выполнения. Значение, указанное в параметре `thisArg`, доступно внутри функции через указатель `this`. В параметре `args` указывается массив с параметрами, передаваемыми функции. Пример использования метода `apply()` совместно с классом `Proxy` приведен в листинге 11.47. Пример:

```
function sum(x, y) {
  console.log(this);
  return x + y;
}
console.log( Reflect.apply(sum, undefined, [30, 10]) ); // 40
// Object [global]
console.log( Reflect.apply(sum, { a: 10 }, [ 15, 6 ]) ); // 21
// { a: 10 }
```

- `construct(target, args[, newTarget])` — возвращает новый объект, вызывая инструкцию `new target(...args)`. В параметре `args` указывается массив с пара-

метрами, передаваемыми конструктору. Прототип функции, указанной в параметре `newTarget`, станет новым прототипом объекта. Внутри конструктора оператор `new.target` будет указывать на `target` или `newTarget`, если параметр `newTarget` указан. Пример использования метода `construct()` совместно с классом `Proxy` приведен в листинге 11.46. Пример:

```
function A(_a, _b) {
  this.a = _a;
  this.b = _b;
  console.log(new.target);
}
function B() {}
let obj = Reflect.construct(A, [10, 20]);
// [Function: A]
console.log( obj );           // A { a: 10, b: 20 }
console.log( obj.__proto__ === A.prototype ); // true
let obj2 = Reflect.construct(A, [30, 40], B);
// [Function: B]
console.log( obj2 );          // B { a: 30, b: 40 }
console.log( obj2.__proto__ === A.prototype ); // false
console.log( obj2.__proto__ === B.prototype ); // true
```

- ❑ `set(target, property, value[, receiver])` — заносит новое значение `value` в свойство `property` объекта `target`. Возвращает значение `true`, если операция успешно выполнена, и `false` — в противном случае. Пример использования метода `set()` совместно с классом `Proxy` приведен в разд. 11.10.2. Пример:

```
let obj = { a: 3 };
console.log( Reflect.set(obj, 'a', 12) ); // true
console.log( Reflect.set(obj, 'b', 88) ); // true
console.log( obj );                       // { a: 12, b: 88 }
```

- ❑ `get(target, property[, receiver])` — возвращает значение свойства `property` объекта `target`. Пример использования метода `get()` совместно с классом `Proxy` приведен в разд. 11.10.2. Пример:

```
let obj = { a: 3 };
console.log( Reflect.get(obj, 'a') ); // 3
console.log( Reflect.get(obj, 'b') ); // undefined
```

- ❑ `has(target, name)` — возвращает значение `true`, если объект `target` содержит свойство `name`, и `false` — в противном случае. Метод является аналогом оператора `in`. Проверяются собственные свойства объекта, а также свойства в цепочке наследования. Пример:

```
let obj = { a: 3 };
console.log( Reflect.has(obj, 'a') );           // true
console.log( Reflect.has(obj, 'b') );           // false
console.log( Reflect.has(obj, 'toString') );    // true
```

- ❑ `deleteProperty(target, property)` — удаляет свойство `property` из объекта `target`. Возвращает значение `true`, если свойство было успешно удалено, и `false` — в противном случае. Если свойство не существует, возвращает `true`. Метод является аналогом оператора `delete` в нестрогом режиме. Пример использования метода `deleteProperty()` совместно с классом `Proxy` приведен в листинге 11.43. Пример:

```
let obj = { a: 3 };
console.log( Reflect.deleteProperty(obj, 'a') ); // true
console.log( Reflect.deleteProperty(obj, 'b') ); // true
console.log( obj );                               // {}
```

В строгом режиме попытка удалить свойство с атрибутом `configurable`, равным `false`, с помощью оператора `delete` приведет к исключению, тогда как метод `deleteProperty()` просто вернет `false`:

```
'use strict';
let obj = Object.freeze({ a: 3 });
console.log( Reflect.deleteProperty(obj, 'a') ); // false
console.log( obj );                               // { a: 3 }
// delete obj.a;
// TypeError: Cannot delete property 'a' of #<Object>
let d = Object.getOwnPropertyDescriptor(obj, 'a');
console.log(d);
// { value: 3, writable: false, enumerable: true,
//   configurable: false }
```

- ❑ `ownKeys(target)` — возвращает массив с именами собственных свойств объекта `target`, включая символные:

```
let obj = Object.create(Object.prototype, {
  'a': { value: 3, enumerable: true },
  'b': { value: 6, enumerable: false },
  [Symbol.for('c')]: { value: 9, enumerable: true }
});
console.log( Reflect.ownKeys(obj) );
// [ 'a', 'b', Symbol(c) ]
```

- ❑ `defineProperty(target, property, attributes)` — аналог одноименного метода из класса `Object` (см. *разд. 11.1.10*). Возвращает значение `true`, если операция успешно выполнена, и `false` — в противном случае. В отличие от метода из класса `Object`, в случае ошибки не генерирует исключение, а просто возвращает `false`. Пример использования метода `defineProperty()` совместно с классом `Proxy` приведен в листинге 11.42. Пример:

```
let obj = Object.create(Object.prototype, {
  'a': {
    value: 10,
    writable: true,
    configurable: false,
```

```

        enumerable: true
    }
});
// Добавление нового свойства
console.log( Reflect.defineProperty(obj, 'b', {
    value: 22,
    writable: true,
    configurable: true,
    enumerable: true
}) ); // true
console.log( obj.b ); // 22
// Попытка изменения атрибутов существующего свойства
// Object.defineProperty(obj, 'a', { enumerable: false });
// TypeError: Cannot redefine property: a
console.log(
    Reflect.defineProperty(obj, 'a', { enumerable: false })
); // false

```

- `getOwnPropertyDescriptor(target, name)` — аналог одноименного метода из класса `Object` (см. *разд. 11.1.9*). Возвращает объект (дескриптор) со всеми атрибутами собственного свойства `name` объекта `target`. Если свойство не найдено, возвращает значение `undefined`. Пример использования метода `getOwnPropertyDescriptor()` совместно с классом `Proxy` приведен в листинге 11.41. Пример:

```

let obj = Object.create(Object.prototype, {
    'a': {
        value: 10,
        writable: true,
        configurable: false,
        enumerable: true
    }
});
console.log( Reflect.getOwnPropertyDescriptor(obj, 'a') );
// { value: 10, writable: true, enumerable: true,
//   configurable: false }
console.log( Reflect.getOwnPropertyDescriptor(obj, 'b') );
// undefined

```

- `setPrototypeOf(target, proto)` — аналог одноименного метода из класса `Object` (см. *разд. 11.1.7*). Возвращает значение `true`, если операция успешно выполнена, и `false` — в противном случае. Пример использования метода `setPrototypeOf()` совместно с классом `Proxy` приведен в листинге 11.45;
- `getPrototypeOf(target)` — аналог одноименного метода из класса `Object` (см. *разд. 11.1.7*). Возвращает ссылку на прототип объекта или значение `null`. Пример использования метода `getPrototypeOf()` совместно с классом `Proxy` приведен в листинге 11.45. Пример:


```

let objA = { a: 10 };
let objB = { b: 55 };
console.log( objA.isPrototypeOf(objB) );           // false
console.log( Reflect.setPrototypeOf(objB, objA) ); // true
console.log( objA.isPrototypeOf(objB) );           // true
console.log(
  Reflect.getPrototypeOf(objA) == Object.prototype
); // true
console.log( Reflect.setPrototypeOf(objA, null) ); // true
console.log(
  Reflect.getPrototypeOf(objA) == Object.prototype
); // false

```

- `preventExtensions(target)` — аналог одноименного метода из класса `Object` (см. *разд. 11.1.11*). Возвращает значение `true`, если операция успешно выполнена, и `false` — в противном случае. Пример использования метода `preventExtensions()` совместно с классом `Proxy` приведен в листинге 11.44;
- `isExtensible(target)` — аналог одноименного метода из класса `Object` (см. *разд. 11.1.11*). Возвращает значение `true`, если объект является расширяемым, и `false` — в противном случае. Пример использования метода `isExtensible()` совместно с классом `Proxy` приведен в листинге 11.44. Пример:

```

let obj = { a: 10 };
console.log( Reflect.isExtensible(obj) );           // true
console.log( Reflect.preventExtensions(obj) );       // true
console.log( Reflect.isExtensible(obj) );           // false

```

11.12. Формат JSON: преобразование объекта в строку и обратно

Простой объект можно преобразовать в формат JSON, а затем выполнить обратное преобразование. При использовании формата JSON следует учитывать следующие моменты:

- данные представляются в строковом формате в кодировке UTF-8;
- имена свойств и значения строкового формата заключаются в двойные кавычки;
- значением свойства может быть только число, строка, массив, объект или `null`;
- конечные запятые не допускаются.

Преобразовать объект в формат JSON позволяет статический метод `stringify()` из класса `JSON`. Формат метода:

```
<Строка> = JSON.stringify(<Объект>[, <Функция или массив>[, <Формат>]])
```

В первом параметре задается объект, преобразуемый в строку в формате JSON:

```

let obj = { n: 5, s: 'строка', b: true, a: [1, 2, 3],
  o: { nb: 32.3 }, nl: null, u: undefined };

```

```
let json = JSON.stringify(obj);
console.log(json);
// {"n":5,"s":"строка","b":true,"a":[1,2,3],
// "o":{"nb":32.3},"nl":null}
console.log(typeof json);
// string
let arr = [1, undefined, 3];
console.log( JSON.stringify(arr) );
// [1,null,3]
```

Если значением свойства является ссылка на функцию, значение типа `symbol` или `undefined`, то такое свойство не попадет в результат. Если значение `undefined` содержится в массиве, оно будет заменено значением `null`.

ОБРАТИТЕ ВНИМАНИЕ

Порядок следования элементов гарантируется только для массивов. Порядок следования свойств может быть произвольным.

Во втором параметре можно указать массив с именами свойств, подлежащих преобразованию:

```
let obj = { id: 10, name: 'Петр', status: true };
let json = JSON.stringify(obj, ['id', 'status']);
console.log(json); // {"id":10,"status":true}
```

Вместо массива можно указать ссылку на функцию, принимающую два параметра: имя свойства и его значение. Функция должна вернуть новое значение, которое и будет кодироваться. Если функция вернет значение `undefined`, то свойство не попадет в итоговую строку. Пример:

```
let obj = { id: 10, name: 'Петр', status: true };
let json = JSON.stringify(obj, function(key, value) {
  console.log(`${key} => ${value}`);
  if (key === 'name') return undefined;
  return value;
});
console.log(json); // {"id":10,"status":true}
```

Обратите внимание на результат вывода в консоль внутри функции:

```
=> [object Object]
id => 10
name => Петр
status => true
```

При первом вызове функции передается ключ с пустой строкой и значением, содержащим ссылку на весь объект. При каждом последующем вызове передается имя конкретного перечислимого свойства и его значение.

Если внутри объекта существует метод `toJSON()`, то будет преобразовываться значение, возвращаемое этим методом, а не сам объект. Создадим внутри метода

`toJSON()` объект, содержащий собственные перечислимые свойства текущего объекта со значениями типа `number`:

```
let obj = { id: 10, status: true, toJSON(n) {
  console.log(typeof n); // string
  let o = {};
  for (const prop in this) {
    if ( this.hasOwnProperty(prop) ) {
      if (typeof this[prop] == 'number')
        o[prop] = this[prop];
    }
  }
  return o;
}};
console.log( JSON.stringify(obj) ); // {"id":10}
```

Метод `toJSON()` реализован в классе `Date`, поэтому значение объекта сразу преобразуется в строку с помощью метода `toISOString()`:

```
let d = new Date(1602708892000);
console.log( d.toJSON() );           // 2020-10-14T20:54:52.000Z
console.log( JSON.stringify(d) );    // "2020-10-14T20:54:52.000Z"
```

ОБРАТИТЕ ВНИМАНИЕ

Преобразуются только собственные перечислимые свойства объекта. Цепочка наследования не просматривается.

Пример:

```
let objA = Object.create(Object.prototype, {
  'a': { value: 10, enumerable: true },
  'b': { value: 5, enumerable: false }
});
let objB = { c: 9 };
Object.setPrototypeOf(objB, objA);
console.log( JSON.stringify(objA) ); // {"a":10}
console.log( JSON.stringify(objB) ); // {"c":9}
```

С помощью метода `toJSON()` мы можем изменить это поведение:

```
let objA = Object.create(Object.prototype, {
  'a': { value: 10, enumerable: true },
  'b': { value: 5, enumerable: false }
});
let objB = { c: 9, toJSON() {
  let o = {};
  for (const prop of Object.getOwnPropertyNames(this)) {
    o[prop] = this[prop];
  }
}}
```

```
let proto = Object.getPrototypeOf(this);
if (!proto || proto == Object.prototype)
    return o;
for ( const prop of Object.getOwnPropertyNames(proto) ) {
    o[prop] = this[prop];
}
return o;
} };
```

```
Object.setPrototypeOf(objB, objA);
console.log( JSON.stringify(objB) ); // {"c":9,"a":10,"b":5}
```

По умолчанию в строке отсутствуют какие-либо разделители между свойствами и значениями, а также отступы. Символы, используемые в качестве отступов, можно указать в параметре `<Формат>`. Укажем три пробела:

```
let obj = { id: 10, name: 'Петр', status: true };
let json = JSON.stringify(obj, null, '   ');
console.log(json);
/*
{
  "id": 10,
  "name": "Петр",
  "status": true
} */
```

Вместо символов можно указать число, означающее количество пробелов. Укажем один пробел:

```
let obj = { id: 10, name: 'Петр', status: true };
let json = JSON.stringify(obj, null, 1);
console.log(json);
/*
{
  "id": 10,
  "name": "Петр",
  "status": true
} */
```

Для обратного преобразования — данных в формате JSON в объект — предназначен статический метод `parse()` из класса `JSON`. Формат метода:

```
<Объект> = JSON.parse(<Данные JSON>[, <Функция>])
```

Если преобразование не удалось, метод генерирует исключение, поэтому преобразование следует выполнять внутри инструкции `try...catch`:

```
let json = '{"id":10,"status":true}';
let obj = null;
try {
    obj = JSON.parse(json);
    console.log(obj);           // { id: 10, status: true }
```

```

    console.log(obj.id);      // 10
    console.log(obj.status); // true
    console.log( JSON.parse('[1,2,3]') ); // [ 1, 2, 3 ]
  } catch (e) {
    console.log('Не удалось декодировать');
  }
}

```

Во втором параметре можно указать ссылку на функцию, в которую будет передаваться имя свойства в виде строки и его значение. Функция должна вернуть значение, которое станет новым значением свойства. Если функция вернет значение `undefined`, свойство не попадет в итоговый объект. Пример:

```

let json = '{"id":10,"status":true}';
let obj = JSON.parse(json, function(key, value) {
  console.log(`${key} => ${value}`);
  if (key === '') return value;
  if (key === 'status') return undefined;
  return value;
});
console.log(obj); // { id: 10 }

```

Обратите внимание на результат вывода в консоль внутри функции:

```

id => 10
status => true
=> [object Object]

```

При каждом вызове передается имя свойства и его значение. При последнем вызове функции передается ключ с пустой строкой и значением, содержащим ссылку на весь объект. Не забудьте для этого случая вернуть объект, иначе результатом работы метода `parse()` станет значение `undefined`.

Для преобразования данных в формате JSON в объект также можно воспользоваться функцией `eval()`, хотя в Node.js лучше ее не использовать. Однако в старых веб-браузерах возможен только такой вариант преобразования, поэтому мы его рассмотрим. Для корректной работы функции `eval()` данные в формате JSON необходимо заключить в круглые скобки. Пример:

```

let json = '{"id":10,"status":true}';
let obj = null;
try {
  obj = eval('(' + json + ')');
  console.log(obj);      // { id: 10, status: true }
  console.log(obj.id);   // 10
  console.log(obj.status); // true
} catch (e) {
  console.log('Не удалось декодировать');
}

```

Если преобразование не удалось, генерируется исключение, поэтому преобразование следует выполнять внутри конструкции `try...catch`, что мы и сделали.



ГЛАВА 12

Модули

Если программа становится слишком большой, ее удобнее разделить на отдельные файлы, называемые *модулями*. Все идентификаторы, объявленные в модуле (переменные, функции, классы), недоступны в других модулях, и чтобы сделать их доступными, необходимо явно выполнить экспорт этих идентификаторов. После чего любой модуль сможет импортировать нужные ему идентификаторы из текущего модуля и использовать их в работе.

Модули можно объединять в *пакеты*. Пакетом называется каталог с модулями, в котором также расположен конфигурационный файл `package.json`.

Node.js поддерживает два типа модулей: CommonJS и ECMAScript. Они довольно заметно отличаются друг от друга.

12.1. Модули CommonJS

Модулем CommonJS по умолчанию является любой программный файл с расширением `js` или `cjs`. Получить ссылку на объект модуля, загруженного самым первым (файл, который является точкой входа в приложение), позволяет свойство `require.main`:

```
console.log(require.main.filename);  
// C:\book\p1\index.js
```

12.1.1. Подключение встроенных модулей

Подключение модулей CommonJS выполняется с помощью функции `require()`:

```
const <Объект> = require(<Модуль>);
```

Если в параметре `<Модуль>` указано только имя модуля, поиск сначала осуществляется среди встроенных модулей. Если `<Модуль>` найден, возвращается объект с экспортированными из модуля идентификаторами. Если такого модуля среди встроенных модулей нет, выполняется поиск в других местах, которые мы рассмотрим далее в этом разделе. Если модуль не найден вообще, генерируется исключение:

```
Error: Cannot find module '...'
```

Подключим модуль `path` и выведем символ-разделитель путей, используемый в текущей операционной системе (содержится в свойстве `delimiter`):

```
const path = require('path');
console.log(path.delimiter);    // В Windows: ;
```

Получим информацию об операционной системе, подключив встроенный модуль `os`:

```
const os = require('os');
console.log( os.platform() );    // win32
console.log( os.version() );    // Windows 10 Pro
console.log( os.type() );        // Windows_NT
```

Внутри одной инструкции можно подключить сразу несколько модулей, указав выражения через запятую:

```
const path = require('path'), os = require('os');
process.stdout.write(path.delimiter + '\n'); // ;
process.stdout.write( os.platform() );        // win32
```

Чтобы получить доступ к отдельным идентификаторам из модуля, следует использовать операцию распаковки объекта (см. *разд. 11.1.3*):

```
const { stdout }    = require('process'),
      { delimiter } = require('path'),
      { platform }  = require('os');
stdout.write(delimiter + '\n');    // ;
stdout.write( platform() );        // win32
```

Встроенные модули Node.js написаны на языке C++ и откомпилированы в машинный код — для повышения производительности.

12.1.2. Получение списка встроенных модулей

Получить список с именами всех встроенных модулей в виде массива строк позволяет свойство `builtinModules` из модуля `module`:

```
console.log(`Версия Node.js: ${process.version}`);
let arrBuiltinModules = require('module').builtinModules;
console.log(arrBuiltinModules);
```

Результат:

```
Версия Node.js: v16.0.0
```

```
[
  '_http_agent',      '_http_client',      '_http_common',
  '_http_incoming',   '_http_outgoing',    '_http_server',
  '_stream_duplex',   '_stream_passthrough', '_stream_readable',
  '_stream_transform', '_stream_wrap',      '_stream_writable',
  '_tls_common',      '_tls_wrap',         'assert',
  'assert/strict',    'async_hooks',       'buffer',
  'child_process',    'cluster',           'console',
```

```

'constants',      'crypto',      'dgram',
'diagnostics_channel', 'dns',      'dns/promises',
'domain',         'events',     'fs',
'fs/promises',    'http',      'http2',
'https',          'inspector', 'module',
'net',            'os',        'path',
'path/posix',     'path/win32', 'perf_hooks',
'process',        'punycode',   'querystring',
'readline',       'repl',       'stream',
'stream/promises', 'string_decoder', 'sys',
'timers',         'timers/promises', 'tls',
'trace_events',   'tty',        'url',
'util',           'util/types', 'v8',
'vm',             'wasi',       'worker_threads',
'zlib'
]

```

Некоторые из перечисленных модулей мы рассмотрим в этой книге, а полное описание всех встроенных модулей можно найти в документации.

12.1.3. Подключение пользовательских модулей

Создадим три файла с именем `module1.js`: первый — в каталоге `C:\book\p1`, второй — во вложенном каталоге `modules`, а третий — в каталоге выше уровнем:

```

C:\book\p1\module1.js
C:\book\p1\modules\module1.js
C:\book\module1.js

```

Содержимое всех трех файлов будет одинаковым:

```

let fname = __filename;
console.log(fname);

```

Чтобы подключить модуль, расположенный на одном уровне вложенности с текущим исполняемым файлом, следует перед именем модуля добавить символы `./`. Если файл с модулем имеет расширение `js`, указывать у него расширение необязательно. Выполним импорт модуля `module1.js` в файле `index.js`, записав в него следующий код:

```
const module1 = require('./module1');
```

Запустим на выполнение файл `index.js`:

```

C:\book\p1>node index.js
C:\book\p1\module1.js

```

Мы успешно получили сообщение из модуля `module1.js`, расположенного в каталоге `C:\book\p1`. Путь к файлу с модулем хранится в переменной `__filename`, которая существует в каждом модуле. Вывод сообщения означает, что модуль успешно загружен, а ссылка на объект модуля сохранена в константе `module1`.

Если же файл с модулем имеет другое расширение (например, `cjs`), расширение при подключении модуля указывать необходимо:

```
const module1 = require('./module1.cjs');
```

Имя константы не обязано совпадать с именем модуля. Можно выбрать любое другое имя, не совпадающее с другими глобальными идентификаторами внутри текущего файла. Например, сохраним объект модуля в константе `m1` и выведем ее значение:

```
const m1 = require('./module1');  
          // C:\book\p1\module1.js  
console.log(m1); // {}
```

Функция `require()` возвращает объект, содержащий все экспортированные из импортируемого модуля идентификаторы. Однако, поскольку мы ничего явно не экспортировали из модуля `module1.js`, то получим пустой объект.

Все идентификаторы — переменные (включая глобальные), функции, классы и пр., — определенные в модуле, видимы только в границах этого модуля (созданная нами в модуле `module1.js` переменная `fname` также видна только в этом модуле). Другой модуль получить к ним доступ не сможет, даже выполнив импорт модуля, в котором они определены. Чтобы разрешить другим модулям пользоваться каким-либо идентификатором, следует явно экспортировать его из модуля (как это сделать, будет рассмотрено далее).

Ранее при импорте мы указали имя файла с модулем без расширения. В этом случае Node.js будет последовательно добавлять к имени файла расширения `js`, `json` и `node` (именно в таком порядке) и проверять, существует ли файл с такими именем и расширением. Если файл существует, он будет загружен и подключен:

```
console.log( require('module')._extensions );  
// [Object: null prototype] {  
//   '.js': [Function],  
//   '.json': [Function],  
//   '.node': [Function]  
// }
```

Расширение `js` обозначает обычный JavaScript-модуль, расширение `node` — модуль, написанный на языке C++ и откомпилированный в машинный код. Расширение `json` обозначает файл в формате JSON, который в результате импорта преобразуется в объект.

Создадим в каталоге `C:\book\p1` файл `myJSON.json` со следующим содержимым:

```
{"id":10,"status":true}
```

Подключим его без указания расширения:

```
const json = require('./myJSON');  
console.log(json);          // { id: 10, status: true }  
console.log(json.id);       // 10  
console.log(json.status);    // true
```

Однако лучше явно указать расширение, чтобы сначала не выполнялся поиск файла `myJSON.js` (если этот файл будет найден, то загружен будет именно он, а не одноименный файл с расширением `json`):

```
const json = require('./myJSON.json');
console.log(json);           // { id: 10, status: true }
```

Если подключаемый модуль расположен во вложенном каталоге, то перед именем модуля указываются символы `./`, после которых следует имя вложенного каталога и символ `/`:

```
const module2 = require('./modules/module1');
```

Запустим на выполнение файл `index.js`:

```
C:\book\pl>node index.js
C:\book\pl\modules\module1.js
```

Для подключения модуля, расположенного в каталоге выше уровнем, перед именем модуля следует указать символы `../`:

```
const module3 = require('../module1');
```

Запустим на выполнение файл `index.js`:

```
C:\book\pl>node index.js
C:\book\module1.js
```

Для каждого выше расположенного каталога нужно дополнительно указывать символы `../`. Например, чтобы выполнить поиск модуля в корне диска, следует указать путь следующим образом:

```
const module4 = require('../../module1');
```

После запуска мы получим сообщение об ошибке:

```
Error: Cannot find module '../../module1'
```

Мы не создавали модуль в корне диска, поэтому и получили исключение. Однако если скопировать файл `module1.js` в корень диска `C:` (для этого нужно обладать правами администратора), то модуль будет успешно загружен:

```
C:\book\pl>node index.js
C:\module1.js
```

Помимо указания относительного пути к модулю можно задать абсолютный путь от корня текущего диска, поставив в начале пути символ `/`:

```
const module1 = require('/book/pl/module1');
const module2 = require('/book/pl/modules/module1');
const module3 = require('/book/module1');
```

Запустим на выполнение файл `index.js`:

```
C:\book\pl>node index.js
C:\book\pl\module1.js
C:\book\pl\modules\module1.js
C:\book\module1.js
```

В операционной системе Windows поиск модулей, пути к которым указаны таким образом, выполняется на текущем диске (так, если текущим является каталог C:\book\p1, поиск модулей будет выполняться на диске C:). Однако можно указать другой диск, поставив его букву и двоеточие в самом начале пути к модулю:

```
const module1 = require('c:/book/p1/module1');
```

Заметьте, что в каждом подключаемом модуле существует одноименная глобальная переменная `fname`, да еще и модули называются одинаково. При этом у нас нет никакого конфликта имен, т. к. каждый модуль имеет свое собственное пространство имен и расположен в отдельном каталоге, а объекты загруженных модулей сохраняются в разных константах.

12.1.4. Кэширование модулей

Все модули после загрузки кэшируются, поэтому при повторном подключении модуля снова загружаться с диска он не будет. Например, изменим содержимое файла `index.js`:

```
const module1 = require('/book/p1/module1');
const module2 = require('/book/p1/module1');
console.log( module1 == module2 );
```

Запустим на выполнение этот файл:

```
C:\book\p1>node index.js
C:\book\p1\module1.js
true
```

Как видно из результата, мы получили только одно сообщение, а две константы ссылаются на один и тот же объект модуля. Однако так может быть не всегда. Например, в операционной системе Windows регистр символов в пути не имеет значения, а вот в индексе кэша — имеет. Указав разный регистр символов в пути, мы получим повторную загрузку модуля и разные объекты модулей. Пример:

```
const module1 = require('/book/p1/module1');
const module2 = require('/BOOK/p1/module1');
console.log( module1 == module2 );
```

Запустим на выполнение файл `index.js`:

```
C:\book\p1>node index.js
C:\book\p1\module1.js
C:\BOOK\p1\module1.js
false
```

Кэшированное содержимое доступно через объект `require.cache`. Запишем в файл `index.js` следующий код:

```
const module1 = require('/book/p1/module1');
console.log( require.cache );
```

Запустим исправленный файл на исполнение:

```
C:\book\p1>node index.js
C:\book\p1\module1.js
[Object: null prototype] {
  'C:\\book\\p1\\index.js': Module {
    id: '.',
    path: 'C:\\book\\p1',
    exports: {},
    parent: null,
    filename: 'C:\\book\\p1\\index.js',
    loaded: false,
    children: [ [Module] ],
    paths: [
      'C:\\book\\p1\\node_modules',
      'C:\\book\\node_modules',
      'C:\\node_modules'
    ]
  },
  'C:\\book\\p1\\module1.js': Module {
    id: 'C:\\book\\p1\\module1.js',
    path: 'C:\\book\\p1',
    exports: {},
    parent: Module {
      id: '.',
      path: 'C:\\book\\p1',
      exports: {},
      parent: null,
      filename: 'C:\\book\\p1\\index.js',
      loaded: false,
      children: [Array],
      paths: [Array]
    },
    filename: 'C:\\book\\p1\\module1.js',
    loaded: true,
    children: [],
    paths: [
      'C:\\book\\p1\\node_modules',
      'C:\\book\\node_modules',
      'C:\\node_modules'
    ]
  }
}
```

Если после загрузки модуля удалить свойство с именем, совпадающим с полным путем к модулю, то загрузка будет выполнена повторно. Запишем в файл `index.js` следующий код:

```
const module1 = require('/book/p1/module1');
const path = require.resolve('/book/p1/module1');
console.log(`path = ${path}`);
delete require.cache[path];
const module2 = require('/book/p1/module1');
console.log( module1 == module2 );
```

В этом примере мы воспользовались методом `resolve()` для получения строки, содержащей полный путь к модулю, который является ключом в объекте кэша.

ОБРАТИТЕ ВНИМАНИЕ

Если модуль не найден, то метод `resolve()` сгенерирует исключение: `Error: Cannot find module '/book/p1/module3'`

Запустим на выполнение исправленный файл `index.js`:

```
C:\book\p1>node index.js
C:\book\p1\module1.js
path = C:\book\p1\module1.js
C:\book\p1\module1.js
false
```

12.1.5. Подключение каталогов

Если при указании относительного пути файл не был найден, то выполняется поиск каталога с указанным именем. Если каталог найден, то сначала проверяется наличие в нем файла `package.json` (т. е. выясняется, является ли каталог пакетом). Если каталог является пакетом, то из файла `package.json` считывается имя модуля (из свойства `main`) и указанный модуль загружается.

Создадим в каталоге `C:\book\p1\modules` файл `package.json` со следующим содержанием:

```
{
  "name": "modules",
  "version": "1.0.0",
  "main": "./module1.js"
}
```

Теперь в файле `C:\book\p1\index.js` подключим каталог:

```
const module1 = require('./modules');
```

Запустим на выполнение файл `index.js`:

```
C:\book\p1>node index.js
C:\book\p1\modules\module1.js
```

Как видно из результата, мы успешно подключили файл `module1.js` из каталога `C:\book\p1\modules`.

Если файл `package.json` в каталоге не найден, или в этом файле отсутствует свойство `main`, или файл с именем, указанным в свойстве `main`, не найден, выполняется попытка загрузки файла с именами `index.js` или `index.node`.

Удалим файл `package.json` из каталога `C:\book\p1\modules`, сделаем копию файла `module1.js` под именем `index.js`, после чего внутри файла `C:\book\p1\index.js` подключим каталог `modules`:

```
const module1 = require('./modules');
```

Запустим файл на выполнение:

```
C:\book\p1>node index.js
```

```
C:\book\p1\modules\index.js
```

Как видно из результата, мы успешно подключили файл `index.js` из каталога `C:\book\p1\modules`.

Если модуль не найден, генерируется исключение:

```
Error: Cannot find module './modules'
```

12.1.6. Подключение модулей и пакетов из каталога *node_modules*

Если имя модуля не начинается с символов `/`, `./` или `../` и не является именем встроенного модуля, поиск выполняется в каталоге `node_modules`, расположенном в том же каталоге, что и исполняемый файл. Если там модуль не будет обнаружен, поиск выполняется в каталоге `node_modules`, расположенном в родительском каталоге, затем — в каталоге, в котором находится родительский каталог, и т. д., пока не будет достигнут корень диска. Если модуль нигде не был найден, генерируется исключение.

Пути поиска можно получить с помощью свойства `paths` объекта `module`:

```
console.log(module.paths);  
/* [  
  'C:\\book\\p1\\node_modules',  
  'C:\\book\\node_modules',  
  'C:\\node_modules'  
] */
```

В *разд. 2.5.2* мы установили пакет `Webpack`. Давайте внутри файла `C:\book\p1\index.js` подключим находящийся в составе этого пакета файл `lib\index.js`, записав путь к нему относительно каталога пакета, и выведем информацию о версии, которая доступна через свойство `version`:

```
const webpack = require('webpack/lib/index.js');  
console.log(webpack.version);  
// 5.36.0
```

Мы можем подключить не отдельный файл, а целый пакет:

```
const webpack = require('webpack');
```

Если пакет установлен глобально, таким способом его не подключить. Каталог с глобальными пакетами не входит в список путей поиска модулей. Получить путь к глобальному каталогу можно так:

```
C:\book\p1>npm root -g
C:\Users\Unicross\AppData\Roaming\npm\node_modules
```

В состав пути входит имя пользователя, поэтому такой путь не будет работать на вашем компьютере — только на компьютере одного из авторов, и только в Windows. В других операционных системах путь может быть иным. Но если мы укажем полный путь и подставим имя "нашего" пользователя, то в Windows все будет работать (пакет sass мы устанавливали глобально в *разд. 2.5.2*):

```
const os = require('os');
let userName = os.userInfo().username;
const sass = require(
  `\\Users\\${userName}\\AppData\\Roaming\\npm\\node_modules\\sass`
);
console.log(sass.info);
// dart-sass      1.32.11 (Sass Compiler) [Dart]
// dart2js 2.12.4  (Dart Compiler) [Dart]
```

Можно получить путь к каталогу с глобальными пакетами динамически, используя метод `execSync()` из пакета `child_process`:

```
let path = require('child_process').execSync('npm root -g');
path = path.toString().trim();
path = path.replace(/\\/g, '/');
const sass = require(`${path}/sass`);
console.log(sass.info);
// dart-sass      1.32.11 (Sass Compiler) [Dart]
// dart2js 2.12.4  (Dart Compiler) [Dart]
```

12.1.7. Переменная окружения **NODE_PATH**

Добавить произвольный каталог в перечень путей поиска модулей можно с помощью установки переменной окружения `NODE_PATH`. Переменная может быть установлена глобально или локально для текущего сеанса работы с приложением Командная строка. Мы рассмотрим второй способ. Для установки переменной окружения в командной строке Windows используется команда:

```
set NODE_PATH=<Путь>
```

Если нужно добавить несколько путей, то они перечисляются через разделитель, принятый в операционной системе (в Windows — точка с запятой, а в UNIX — двоеточие). Добавим путь к глобальным пакетам и к каталогу `C:\book\p1\modules`:

```
C:\book\p1>set NODE_PATH=C:\Users\%UserName%\AppData\Roaming\npm\node_modules;C:\book\p1\modules
```

Все добавленные пути можно увидеть через свойство `globalPaths` объекта встроенного модуля `module`:

```
console.log( require('module').globalPaths );
/* [
  'C:\\Users\\Unicross\\AppData\\Roaming\\npm\\node_modules',
```

```
'C:\\book\\p1\\modules',  
'C:\\Users\\Unicross\\.node_modules',  
'C:\\Users\\Unicross\\.node_modules\\node_modules',  
'C:\\nodejs\\node16\\lib\\node'  
] */
```

Теперь мы можем подключить глобальный пакет sass без указания пути к нему (пакет sass мы устанавливали глобально в *разд. 2.5.2*):

```
const sass = require('sass');  
console.log(sass.info);  
// dart-sass      1.32.11 (Sass Compiler) [Dart]  
// dart2js 2.12.4  (Dart Compiler) [Dart]
```

Аналогичным образом можно подключить модуль module1.js из каталога C:\book\p1\modules:

```
const m = require('module1.js');
```

Запустим на выполнение файл index.js:

```
C:\book\p1>node index.js  
C:\book\p1\modules\module1.js
```

Чтобы подключить одноименный модуль из другого каталога, следует указать относительный путь:

```
const m = require('./module1.js');  
// C:\book\p1\module1.js
```

12.1.8. Экспорт идентификаторов из модуля

До этого момента мы подключали пользовательские модули, внутри которых просто выводили сообщение в консоль, чтобы было видно, что модуль загружается. При этом мы получали пустой объект модуля. Пришла пора научиться экспортировать из модуля идентификаторы.

Перед запуском модуля его код оборачивается следующей функцией:

```
(function(exports, require, module, __filename, __dirname) {  
  // Код модуля  
  return module.exports;  
})(/* Значения */);
```

Теперь понятно, почему область видимости глобальных переменных ограничена файлом, внутри которых они объявлены? Ведь переменная, объявленная внутри функции, является локальной, и видима она только внутри функции.

Через переменную __dirname доступен абсолютный путь к каталогу с модулем, а через переменную __filename — абсолютный путь к файлу модуля. Выведем значения переменных внутри файла C:\book\p1\module1.js:

```
console.log(__dirname); // C:\book\p1  
console.log(__filename); // C:\book\p1\module1.js
```


При этом мы запускаем файл `C:\book\p1\index.js`, содержащий следующий код:

```
const module1 = require('./module1');  
console.log(module1);
```

Через переменную `module` доступен объект модуля. Значение свойства `exports` этого объекта возвращается в качестве результата функцией `require()` после загрузки модуля. Выведем значение этого свойства внутри файла `C:\book\p1\module1.js`:

```
console.log(module.exports); // {}
```

По умолчанию свойство `module.exports` содержит пустой объект, в который мы можем добавлять свойства и методы. Также можно присвоить этому свойству другое значение. Например, запишем в него число, которое получим в основном файле:

```
module.exports = 12;
```

Запустим на выполнение файл `index.js`:

```
C:\book\p1>node index.js  
12
```

Как видно из результата, мы получили число 12 после подключения модуля. Вместо присваивания числа свойству `module.exports` можно добавить свойство в объект:

```
module.exports.x = 12;  
module.exports.y = 5;
```

Запустим на выполнение файл `index.js`:

```
C:\book\p1>node index.js  
{ x: 12, y: 5 }
```

Таким образом, добавляя свойства в объект `module.exports`, можно экспортировать идентификаторы из модуля и сделать их доступными внутри основного файла через результат выполнения функции `require()`. Но помните, что это необязательно должен быть объект по умолчанию. Можно присвоить другое значение, например число, строку, ссылку на функцию или объект класса. Пример:

```
class Car {  
  constructor(m) {  
    this.model = m;  
  }  
  getModel() {  
    return this.model;  
  }  
}  
module.exports.car = new Car('Audi');
```

Запустим на выполнение файл `index.js`:

```
C:\book\p1>node index.js  
{ car: Car { model: 'Audi' } }
```

Переменная `exports` хранит ссылку на объект, изначально присвоенный свойству `module.exports`. Эту переменную можно использовать вместо свойства `module.exports`, сокращая объем кода. Запишем в файл `index.js` следующий код:

```
exports.x = 12;
exports.y = 5;
```

Запустим исправленный файл на выполнение:

```
C:\book\p1>node index.js
{ x: 12, y: 5 }
```

Помните, что переменная `exports` ссылается на объект внутри свойства `module.exports` только до момента присваивания значения. Причем возвращается значение свойства `module.exports`, а не переменной `exports`. Для примера запишем в файл `index.js` следующий код:

```
exports = { x: 12 }; // Не экспортируется!
module.exports.y = 5;
```

Запустим на выполнение исправленный файл:

```
C:\book\p1>node index.js
{ y: 5 }
```

Поэтому присваивание нового объекта нужно выполнять так:

```
module.exports = exports = { x: 12, y: 5 };
exports.z = 8;
```

После запуска файла на выполнение получим вот что:

```
{ x: 12, y: 5, z: 8 }
```

12.1.9. Объект *module*

Через переменную `module` внутри каждого файла доступен объект текущего модуля. Выведем содержимое объекта внутри файла `index.js`:

```
console.log(module);
```

Результат:

```
Module {
  id: '.',
  path: 'C:\\book\\p1',
  exports: {},
  parent: null,
  filename: 'C:\\book\\p1\\index.js',
  loaded: false,
  children: [],
  paths: [
    'C:\\book\\p1\\node_modules',
    'C:\\book\\node_modules',
    'C:\\node_modules'
  ]
}
```

Объект `module` содержит следующие основные свойства:

- ❑ `id` — идентификатор модуля (например: `'.'` или `'C:\\book\\p1\\module1.js'`);
- ❑ `filename` — абсолютный путь к файлу модуля (например: `'C:\\book\\p1\\index.js'`);
- ❑ `path` — абсолютный путь к каталогу, в котором находится файл модуля (например: `'C:\\book\\p1'`);
- ❑ `paths` — массив с путями поиска модулей;
- ❑ `loaded` — `true`, если модуль уже загружен, и `false`, если он загружается в данный момент;
- ❑ `exports` — объект с экспортируемыми идентификаторами;
- ❑ `children` — массив с объектами модулей, загруженными текущим модулем, или пустой массив.

Свойство `require.main` содержит ссылку на объект модуля, являющегося точкой входа. Проверить, является ли текущий файл точкой входа, можно так:

```
require('./module1');
console.log(require.main === module);
// true (внутри C:\\book\\p1\\index.js)
// false (внутри C:\\book\\p1\\module1.js)
console.log(require.main.filename);
// C:\\book\\p1\\index.js
```

12.2. Модули ECMAScript (ESM)

Стандарт ECMAScript предлагает новый формат JavaScript-модулей.

12.2.1. Основные отличия модулей ECMAScript от модулей CommonJS

Прежде чем изучать модули ECMAScript, создадим новый пакет с именем `p2` (`C:\\book\\p2`) и добавим в него файл `package.json` со следующим содержимым:

```
{
  "name": "p2",
  "version": "1.0.0",
  "main": "app.mjs",
  "type": "commonjs",
  "scripts": {
    "test": "node app.mjs"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": ""
}
```

Обратите внимание на две особенности. Во-первых, на значение свойства `main`. Мы указали расширение `mjs`, а не `js`. По умолчанию расширение `js` связано с модулями CommonJS. Во-вторых, чтобы явно указать это, мы добавили свойство `type` со значением `"commonjs"`. Если нужно связать расширение `js` с модулями ECMAScript, то свойству `type` следует дать значение `"module"`. На данном этапе мы оставим связь расширения `js` с модулями CommonJS, а модули ECMAScript будем хранить в файлах с расширением `mjs`. Кстати, файлы с расширением `cjs` всегда являются модулями CommonJS.

Модули ECMAScript и CommonJS подключаются по-разному. Кроме того, пути, указанные в переменной окружения `NODE_PATH`, при поиске модуля ECMAScript не просматриваются.

НА ЗАМЕТКУ

Код модулей ECMAScript всегда выполняется в строгом режиме.

12.2.2. Подключение встроенных модулей

Подключение модулей ECMAScript выполняется с помощью инструкции `import`:

```
import <Модуль>;
import { <Идентификатор 1>[ as <Псевдоним 1>][, ...,
    <Идентификатор N>[ as <Псевдоним N>]] } from <Модуль>;
import * as <Объект> from <Модуль>;
```

Первый формат инструкции `import` просто загружает и выполняет модуль без импортирования идентификаторов. Второй формат импортирует идентификаторы, указанные внутри фигурных скобок. После ключевого слова `as` можно задать псевдоним, который будет использоваться вместо идентификатора. Третий формат помещает все идентификаторы, экспортированные внутри модуля, в `<Объект>`.

Если в параметре `<Модуль>` указано только имя модуля, поиск сначала осуществляется среди встроенных модулей. Если такого модуля среди встроенных модулей нет, то выполняется поиск в других местах, которые мы рассмотрим далее в этом разделе. Если модуль не найден вообще, то генерируется исключение:

```
Error [ERR_MODULE_NOT_FOUND]: Cannot find package '...' imported from 
C:\book\p2\app.mjs
```

Создадим точку входа в приложение — файл `app.mjs` (`C:\book\p2\app.mjs`), — подключим встроенный модуль `process` и импортируем объект `stdout`:

```
import { stdout } from 'process';
stdout.write('Строка'); // Строка
```

Таким же способом можно подключить любой другой встроенный модуль. Например, подключим модуль `path` и импортируем идентификаторы `delimiter` и `sep`:

```
import { delimiter, sep } from 'path';
console.log(delimiter); // B Windows: ;
console.log(sep); // B Windows: \
```

Если какой-либо импортируемый идентификатор конфликтует с одноименным идентификатором внутри текущего файла, то достаточно после ключевого слова `as` задать его псевдоним. Использовать идентификатор после указания псевдонима нельзя. Пример:

```
import { delimiter as d, sep } from 'path';
console.log(d);           // B Windows: ;
console.log(sep);         // B Windows: \
// console.log(delimiter);
// ReferenceError: delimiter is not defined
```

Можно также импортировать все идентификаторы и поместить их в объект с уникальным именем. Получим информацию об операционной системе, подключив встроенный модуль `os`:

```
import * as os from 'os';
console.log( os.platform() ); // win32
console.log( os.version() );  // Windows 10 Pro
console.log( os.type() );     // Windows_NT
```

Для быстрой вставки инструкции `import` в редакторе VS Code применяется следующая комбинация:

```
import statement =>
import {  } from "module";
```

12.2.3. Получение списка встроенных модулей

Получить список с именами всех встроенных модулей в виде массива строк позволяет свойство `builtinModules` из модуля `module`:

```
import { version } from 'process';
import { builtinModules } from 'module';
console.log(`Версия Node.js: ${version}`);
console.log(builtinModules);
```

Результат:

```
Версия Node.js: v16.0.0
```

```
[
  '_http_agent',      '_http_client',      '_http_common',
  '_http_incoming',   '_http_outgoing',    '_http_server',
  '_stream_duplex',   '_stream_passthrough', '_stream_readable',
  '_stream_transform', '_stream_wrap',      '_stream_writable',
  '_tls_common',      '_tls_wrap',         'assert',
  'assert/strict',    'async_hooks',       'buffer',
  'child_process',    'cluster',           'console',
  'constants',        'crypto',            'dgram',
  'diagnostics_channel', 'dns',               'dns/promises',
  'domain',           'events',            'fs',
```

```

'fs/promises',      'http',      'http2',
'https',            'inspector', 'module',
'net',              'os',        'path',
'path/posix',       'path/win32', 'perf_hooks',
'process',          'punycode',  'querystring',
'readline',         'repl',      'stream',
'stream/promises',  'string_decoder', 'sys',
'timers',           'timers/promises', 'tls',
'trace_events',     'tty',       'url',
'util',             'util/types', 'v8',
'vm',               'worker_threads', 'zlib'
]

```

12.2.4. Подключение пользовательских модулей

Создадим три файла с именем `module1.mjs`: первый — в каталоге `C:\book\p2`, второй — во вложенном каталоге `modules`, а третий — в каталоге выше уровнем:

```

C:\book\p2\module1.mjs
C:\book\p2\modules\module1.mjs
C:\book\module1.mjs

```

Содержимое всех трех файлов будет одинаковым:

```

let fname = import.meta.url;
console.log(fname);

```

Через свойство `import.meta.url`, существующее в каждом модуле, доступен URL с полным путем к текущему файлу и его именем, например:

```
file:///C:/book/p2/module1.mjs
```

Чтобы подключить модуль, расположенный на одном уровне вложенности с текущим исполняемым файлом, следует перед именем модуля добавить символы `./`. Кроме того, нужно явным образом указать расширение файла с модулем ECMAScript. Запишем в файл `app.mjs` код, подключающий модуль `module1.mjs`:

```

import * as m from './module1.mjs';
console.log(m);

```

Запустим на выполнение файл `app.mjs`:

```

C:\book\p2>node app.mjs
file:///C:/book/p2/module1.mjs
[Module: null prototype] { }

```

Как видно из результата, модуль `module1.mjs` из каталога `C:\book\p2` был успешно загружен, выполнен и вывел сообщение. Ссылка на объект с импортированными из этого модуля идентификаторами сохранена в константе `m`. Сейчас объект пуст, т. к. ни один идентификатор из модуля экспортирован не был. Как выполнить экспорт, мы рассмотрим далее в этой главе.

Учитывая, что нам нужно просто загрузить модуль и внутри него вывести сообщение в консоль, импорт можно выполнить так:

```
import './module1.mjs';
```

Если подключаемый модуль расположен во вложенном каталоге, то перед именем модуля указываются символы `./`, после которых следует имя вложенного каталога и символ `/`:

```
import './modules/module1.mjs';
```

Запустим на выполнение файл `app.mjs`:

```
C:\book\p2>node app.mjs
file:///C:/book/p2/modules/module1.mjs
```

Для подключения модуля, расположенного в каталоге выше уровнем, перед именем модуля следует указать символы `../`:

```
import '../module1.mjs';
```

Запустим на выполнение файл `app.mjs`:

```
C:\book\p2>node app.mjs
file:///C:/book/module1.mjs
```

Для каждого выше расположенного каталога нужно дополнительно указывать символы `../`. Например, чтобы выполнить поиск модуля в корне диска, следует указать путь следующим образом:

```
import '../../module1.mjs';
```

После запуска мы получим сообщение об ошибке, поскольку такой модуль не существует:

```
Error [ERR_MODULE_NOT_FOUND]: Cannot find module 'C:\module1.mjs' ↵
```

```
imported from C:\book\p2\app.mjs
```

Однако если скопировать файл `module1.mjs` в корень диска `C:` (для этого нужно обладать правами администратора), модуль будет успешно загружен:

```
C:\book\p2>node app.mjs
file:///C:/module1.mjs
```

Можно задать абсолютный путь к модулю, записав его в виде URL с протоколом `file:///`:

```
import * as module1 from 'file:///C:/book/p2/module1.mjs';
import * as module2 from 'file:///C:/book/p2/modules/module1.mjs';
import * as module3 from 'file:///C:/book/module1.mjs';
```

Запустим на выполнение файл `app.mjs`:

```
C:\book\p2>node app.mjs
file:///C:/book/p2/module1.mjs
file:///C:/book/p2/modules/module1.mjs
file:///C:/book/module1.mjs
```

Заметьте, что в каждом подключаемом модуле существует своя глобальная переменная `fname`, да еще и модули называются одинаково. При этом у нас нет никакого конфликта имен, т. к. каждый модуль имеет свое собственное пространство имен и расположен в отдельном каталоге, а объекты загруженных модулей сохраняются в разных константах.

12.2.5. Кэширование модулей

Следует учитывать, что модули после загрузки кэшируются, поэтому при повторном подключении модуля дополнительной загрузки не будет:

```
import * as module1 from './module1.mjs';
import * as module2 from './module1.mjs';
console.log( module1 == module2 );
```

Запустим на выполнение файл `app.mjs`:

```
C:\book\p2>node app.mjs
file:///C:/book/p2/module1.mjs
true
```

Мы получили только одно сообщение, а две константы ссылаются на один и тот же объект модуля. Однако так может быть не всегда. Например, в операционной системе Windows регистр символов в пути не имеет значения, а вот в индексе кэша важен. Указав разный регистр символов в пути, мы получим повторную загрузку модуля и разные объекты модулей. Пример:

```
import * as module1 from './module1.mjs';
import * as module2 from './MODULE1.mjs';
console.log( module1 == module2 );
```

Запустим на выполнение файл `app.mjs`:

```
C:\book\p2>node app.mjs
file:///C:/book/p2/module1.mjs
file:///C:/book/p2/MODULE1.mjs
false
```

Если после расширения файла добавить строку GET-параметров, загрузка будет выполнена повторно. Внесем в файл `app.mjs` следующий код:

```
import * as module1 from './module1.mjs';
import * as module2 from './module1.mjs?_=12345';
console.log( module1 == module2 );
```

Запустим на выполнение исправленный файл:

```
C:\book\p2>node app.mjs
file:///C:/book/p2/module1.mjs
file:///C:/book/p2/module1.mjs?_=12345
false
```

ПРИМЕЧАНИЕ

Для кэширования модулей CommonJS и ECMAScript используются разные кэши.

12.2.6. Подключение модулей и пакетов из каталога *node_modules*

Если имя модуля не начинается с символов `./` или `../` и не является именем встроенного модуля, поиск выполняется в каталоге `node_modules`, расположенном в том же каталоге, что и исполняемый файл. Если там модуль не будет обнаружен, поиск выполняется в каталоге `node_modules`, расположенном в родительском каталоге, затем — в каталоге, в котором находится родительский каталог, и т. д., пока не будет достигнут корень диска. Если модуль нигде не найден, генерируется исключение.

Создадим каталог `C:\book\p2\node_modules\myPackage`, а в нем — файл `package.json` со следующим содержимым:

```
{
  "name": "myPackage",
  "version": "1.0.0",
  "main": "main.mjs",
  "type": "commonjs"
}
```

Добавим в тот же каталог файл `main.mjs` со следующим содержимым:

```
console.log(import.meta.url);
```

Если при подключении указано только имя пакета, то выполняется поиск файла `package.json` и используется значение его свойства `main`. Подключим наш пакет `myPackage`, записав в файл `app.mjs` следующий код:

```
import * as obj from 'myPackage';
```

Запустим на выполнение файл `app.mjs`:

```
C:\book\p2>node app.mjs
file:///C:/book/p2/node_modules/myPackage/main.mjs
```

Также можно подключить конкретный файл из пакета, указав путь к нему относительно каталога пакета и его имя с расширением. Исправим код в файле `app.mjs` следующим образом:

```
import * as obj from 'myPackage/main.mjs';
```

Запустим на выполнение исправленный файл:

```
C:\book\p2>node app.mjs
file:///C:/book/p2/node_modules/myPackage/main.mjs
```

Если пакет установлен глобально, то таким способом его не подключить. Каталог с глобальными пакетами не входит в пути поиска модулей.

12.2.7. Экспорт идентификаторов из модуля

Экспорт идентификаторов из модуля выполняется с помощью инструкции `export`. Основные форматы инструкции:

```
export <Объявление идентификатора>;  
export { <Идентификатор 1>[ as <Псевдоним 1>][, ...,  
    <Идентификатор N>[ as <Псевдоним N>]] };
```

Первый формат можно использовать при объявлении переменных, функций, объектов, классов и т. д. Запишем в файл `C:\book\p2\module1.mjs` код из листинга 12.1.

Листинг 12.1. Первый формат инструкции `export`

```
export let n = 10;  
export const str = 'string';  
export const obj = { c: 20 };  
export function print(msg) {  
    console.log(msg);  
}  
export class Car {  
    constructor(m) {  
        this.model = m;  
    }  
    getModel() {  
        return this.model;  
    }  
}
```

Подключим модуль внутри файла `app.mjs` и выведем значения импортированных идентификаторов:

```
import { n, str as s, obj, print, Car } from './module1.mjs';  
console.log(n);           // 10  
console.log(s);           // string  
console.log(obj);         // { c: 20 }  
let car = new Car('Audi');  
print( car.getModel() );  // Audi
```

Следует учитывать, что все импортированные идентификаторы являются константами, независимо от способа объявления внутри модуля. Например, переменная `n` внутри модуля объявлена с помощью ключевого слова `let`, но после импорта в коде импортировавшего ее файла она является константой, и попытка изменения ее значения приведет к исключению:

```
n = 2;  
// TypeError: Assignment to constant variable.
```

Однако мы можем менять значения свойств внутри импортированного объекта:

```
obj.c = 5;  
console.log(obj);           // { c: 5 }
```

Необязательно импортировать все идентификаторы из модуля — достаточно указать только идентификаторы, которые нам нужны. Импортируем класс `Car` и функцию `print()` для которой создадим псевдоним:

```
import { print as echo, Car } from './module1.mjs';
let car = new Car('Kia');
echo( car.getModel() );    // Kia
```

Если требуется много идентификаторов, удобнее импортировать их все в виде объекта, а затем обращаться к идентификаторам как к свойствам этого объекта:

```
import * as m from './module1.mjs';
console.log(m.n);           // 10
console.log(m.str);         // string
console.log(m.obj);         // { c: 20 }
let car = new m.Car('Audi');
m.print( car.getModel() );  // Audi
```

Второй формат инструкции `export` позволяет экспортировать идентификаторы уже после объявления. В этом случае идентификаторы перечисляются через запятую внутри фигурных скобок. Запишем в файл `C:\book\p2\module1.mjs` код из листинга 12.2.

Листинг 12.2. Второй формат инструкции `export`

```
let n = 10;
const str = 'string';
const obj = { c: 20 };
function print(msg) {
    console.log(msg);
}
class Car {
    constructor(m) {
        this.model = m;
    }
    getModel() {
        return this.model;
    }
}
export { n, str, obj, print, Car };
```

Импортируем модуль:

```
import * as m from './module1.mjs';
console.log(m);
/*
[Module: null prototype] {
  Car: [class Car],
  n: 10,
  obj: { c: 20 },
  print: [Function: print],
  str: 'string'
} */
```

У любого экспортируемого идентификатора можно указать псевдоним, записав его после ключевого слова `as`:

```
export { n as x, str as s, obj as o, print as echo, Car };
```

Импортируем модуль (заметьте, что имена импортируемых идентификаторов заменились псевдонимами):

```
import * as m from './module1.mjs';
console.log(m);
/*
[Module: null prototype] {
  Car: [class Car],
  echo: [Function: print],
  o: { c: 20 },
  s: 'string',
  x: 10
} */
```

12.2.8. Экспорт идентификатора по умолчанию

Внутри модуля один идентификатор можно объявить экспортируемым по умолчанию. Для этого предназначены следующие форматы инструкции `export`:

```
export default <Выражение, ссылка на функцию или класс>;
export default <Объявление идентификатора по умолчанию>;
export { <Идентификатор по умолчанию> as default,
        <Идентификатор 1>[ as <Псевдоним 1>][, ...,
        <Идентификатор N>[ as <Псевдоним N>]] };
```

Первый формат задает выражение, используемое в качестве значения идентификатора по умолчанию:

```
export default { c: 20 };
```

Можно также указать ссылку на анонимную функцию:

```
export default function(msg) {
  console.log(msg);
}
```

Или ссылку на анонимный класс:

```
export default class {
  constructor(m) {
    this.model = m;
  }
}
```

Второй формат можно использовать при объявлении функции или класса. Запишем в файл `C:\book\p2\module1.mjs` код из листинга 12.3.

Листинг 12.3. Второй формат инструкции `export default`

```
const obj = { c: 20 };
function print(msg) {
  console.log(msg);
}
export default class Car {
  constructor(m) {
    this.model = m;
  }
  getModel() {
    return this.model;
  }
}
export { obj, print };
```

ОБРАТИТЕ ВНИМАНИЕ

Только один идентификатор можно объявить экспортируемым по умолчанию.

Если объявить несколько идентификаторов, то генерируется исключение:

```
SyntaxError: Duplicate export of 'default'
```

Теперь импортируем модуль:

```
import * as m from './module1.mjs';
console.log(m);
/*
[Module: null prototype] {
  default: [class Car],
  obj: { c: 20 },
  print: [Function: print]
} */
```

Свойство `default` обозначает идентификатор, экспортируемый по умолчанию.

С помощью третьего формата инструкции `export` можно самим обозначить идентификатор по умолчанию с помощью псевдонима `default` (листинг 12.4).

Листинг 12.4. Использование инструкции `export` для экспорта по умолчанию

```
const obj = { c: 20 };
function print(msg) {
  console.log(msg);
}
class Car {
  constructor(m) {
    this.model = m;
  }
}
```

```
getModel() {  
    return this.model;  
}  
}  
export { obj, print, Car as default };
```

При подключении модуля можно использовать следующие форматы инструкции `import`:

```
import <Идентификатор по умолчанию> from <Модуль>;  
import <Идентификатор по умолчанию>,  
    { <Идентификатор 1>[ as <Псевдоним 1>][, ...,  
        <Идентификатор N>[ as <Псевдоним N>]] } from <Модуль>;  
import <Идентификатор по умолчанию>, * as <Объект> from <Модуль>;
```

Первый формат позволяет импортировать только идентификатор по умолчанию:

```
import Car from './module1.mjs';  
console.log( new Car('Audi') );    // Car { model: 'Audi' }
```

Причем идентификатор по умолчанию мы можем сразу переименовать, если его имя совпадает с именем другого идентификатора в программе:

```
import MyCar from './module1.mjs';  
console.log( new MyCar('Audi') );  
// Car { model: 'Audi' }
```

Второй формат позволяет дополнительно импортировать другие идентификаторы:

```
import MyCar, { obj, print as echo } from './module1.mjs';  
echo( new MyCar('Audi') );    // Car { model: 'Audi' }  
echo( obj );                  // { c: 20 }
```

При использовании третьего формата импортируются все идентификаторы:

```
import Car, * as m from './module1.mjs';  
m.print( new Car('Audi') );    // Car { model: 'Audi' }  
m.print( m.obj );              // { c: 20 }
```

12.2.9. Резэкспорт

Часто в модуле импортируются какие-либо идентификаторы из других модулей с целью тут же экспортировать их (выполнить резэкспорт). Плюс такого подхода заключается в уменьшении количества инструкций `import` в основном файле. С помощью следующих форматов инструкции `export` можно реализовать резэкспорт идентификаторов из модулей:

```
export { <Идентификатор 1>[ as <Псевдоним 1>][, ...,  
    <Идентификатор N>[ as <Псевдоним N>]] } from <Модуль>;  
export * from <Модуль>;
```

Первый формат загружает модуль `<Модуль>` и резэкспортирует идентификаторы, перечисленные через запятую внутри фигурных скобок. Любому резэкспортируемому

идентификатору можно дать произвольный псевдоним, указав его после ключевого слова `as`. Второй формат загружает модуль `<Модуль>` и реэкспортирует сразу все идентификаторы, кроме идентификаторов по умолчанию.

ОБРАТИТЕ ВНИМАНИЕ

Внутри текущего модуля нет доступа к реэкспортируемым идентификаторам!

Создадим внутри каталога `C:\book\p2\modules` файлы `module1.mjs`, `module2.mjs`, `module3.mjs` и `allModules.mjs`. Содержимое файла `module1.mjs`:

```
let n = 10;
function f1() {
  console.log(`f1() - ${import.meta.url}`);
}
export { n, f1 };
```

Содержимое файла `module2.mjs`:

```
export const str = 'string';
export function f2() {
  console.log(`f2() - ${import.meta.url}`);
}
```

Содержимое файла `module3.mjs`:

```
const o = { c: 20 };
export function f3() {
  console.log(`f3() - ${import.meta.url}`);
}
export { o };
```

Содержимое файла `allModules.mjs`:

```
export { n, f1 } from './module1.mjs';
export { str as s, f2 } from './module2.mjs';
export * from './module3.mjs';
// К этим идентификаторам в текущем модуле доступа нет!
// console.log(n);
// ReferenceError: n is not defined
```

Теперь подключим модуль `allModules.mjs` в основном файле и обратимся к импортированным идентификаторам из всех модулей:

```
import * as obj from './modules/allModules.mjs';
console.log(obj.n); // 10
console.log(obj.s); // string
console.log(obj.o); // { c: 20 }
obj.f1();           // f1() - file:///C:/book/p2/modules/module1.mjs
obj.f2();           // f2() - file:///C:/book/p2/modules/module2.mjs
obj.f3();           // f3() - file:///C:/book/p2/modules/module3.mjs
```

При использовании реэкспорта следует учитывать особенности работы с идентификаторами по умолчанию. Изменим код файла `C:\book\p2\modules\module1.mjs` следующим образом:

```
export const PI = 3.14;
export default class Car {
  constructor(m) {
    this.model = m;
  }
}
```

Здесь мы экспортируем константу и класс, последний — как идентификатор по умолчанию. Теперь попробуем выполнить реэкспорт, изменив содержимое файла `C:\book\p2\modules\allModules.mjs` следующим образом:

```
export * from './module1.mjs';
```

Подключим этот модуль и посмотрим на объект модуля внутри файла `C:\book\p2\app.mjs`:

```
import * as obj from './modules/allModules.mjs';
console.log(obj); // [Module] { PI: 3.14 }
```

Внутри объекта модуля имеется только константа, а свойство `default` отсутствует. Таким образом, идентификатор по умолчанию "потерялся". Чтобы это исправить, внутри файла `allModules.mjs` мы должны указать `default` явным образом, например так:

```
export * from './module1.mjs';
export { default } from './module1.mjs';
```

или так:

```
export { PI, default } from './module1.mjs';
```

Подключим модуль и посмотрим на объект модуля внутри файла `app.mjs`:

```
import * as obj from './modules/allModules.mjs';
console.log(obj); // [Module] { PI: 3.14, default: [class Car] }
```

Теперь все в порядке, идентификатор по умолчанию на месте.

Можно также сделать идентификатор по умолчанию обычным именованным идентификатором, указав новое имя после ключевого слова `as`:

```
export { PI, default as Car } from './module1.mjs';
```

Подключим модуль и посмотрим на объект модуля внутри файла `app.mjs`:

```
import * as obj from './modules/allModules.mjs';
console.log(obj); // [Module] { Car: [class Car], PI: 3.14 }
```

12.2.10. Динамический импорт

Инструкцию `import` нельзя вложить в какой-либо блок. Это означает, что мы не можем использовать ее внутри оператора ветвления, чтобы подключить тот или иной модуль в зависимости от какого-либо условия. Мы даже не можем обработать исключение, генерируемое, если подключаемый модуль не существует. Более того,

значение, указанное после ключевого слова `from`, должно быть строкой внутри одинарных или двойных кавычек. Попытка подставить переменную приведет к исключению, например:

```
let path = './modules/module1.mjs';
import * as obj from path;
// SyntaxError: Unexpected identifier
```

Попытка выполнить конкатенацию строк также приведет к исключению:

```
import * as obj from './modules/' + 'module1.mjs';
// SyntaxError: Unexpected token '+'
```

Строки в обратных кавычках также недопустимы:

```
import * as obj from `./modules/module1.mjs`;
// SyntaxError: Unexpected template string
```

Для динамического импорта предусмотрен особый формат инструкции `import`:

```
import (<Модуль>)
```

Во-первых, это не функция. Во-вторых, динамическая загрузка модуля выполняется асинхронно. Инструкция `import()` возвращает объект класса `Promise`, с помощью которого можно обработать результат загрузки. Внутри функции, назначенной с помощью метода `then()`, через параметр доступен объект модуля при его успешной загрузке. Внутри функции, назначенной с помощью метода `catch()`, через параметр доступно описание ошибки при неудачной загрузке модуля. Пример динамического импорта:

```
let name = 'module1.mjs';
import(`./modules/${name}`).then(
  function(m) {
    console.log(m);
  }).catch( function(e) {
    console.log('Не удалось загрузить модуль');
  });
console.log('Динамический импорт выполняется асинхронно');
```

Поскольку динамический импорт выполняется асинхронно, в консоли мы получим следующую последовательность вывода:

```
C:\book\p2>node app.mjs
Динамический импорт выполняется асинхронно
[Module] { f1: [Function: f1], n: 10 }
```

Сначала мы получили результат вывода после инструкции `import()`, а лишь затем выполнен код внутри функции, назначенной с помощью метода `then()` объекта класса `Promise`. Весь код, использующий идентификаторы из загруженного модуля, должен быть расположен внутри функции, что часто очень неудобно. Решение данной проблемы — применение ключевых слов `async` и `await`. Пример приведен в листинге 12.5.

Листинг 12.5. Динамический импорт с применением инструкций `async` и `await`

```
(async function() {
  let modules = ['./modules/module1.mjs', './modules/module2.mjs'];
  let m = {};
  try {
    let obj;
    for (const module of modules) {
      obj = await import(module);
      Object.assign(m, obj);
    }
  } catch (e) {
    console.log('Не удалось загрузить модуль.', e);
  }
  console.log(m);
  console.log('Динамический импорт выполняется асинхронно');
})();
```

Следует учитывать, что при использовании этого кода значения одноименных идентификаторов будут перезаписаны. Это также касается идентификатора по умолчанию, значение которого внутри объекта модуля записывается в свойство `default`. Решение данной проблемы — сохранение объектов модулей в переменных, массиве или словаре по отдельности.

Теперь последовательность выполнения инструкций будет правильной:

```
C:\book\p2>node app.mjs
{ f1: [Function: f1], n: 10, f2: [Function: f2], str: 'string' }
Динамический импорт выполняется асинхронно
```

ПРИМЕЧАНИЕ

Подробно инструкцию `try...catch` и асинхронность мы рассмотрим в последующих главах.

12.2.11. Подключение модулей CommonJS

Инструкцию `import` можно использовать и для подключения модулей CommonJS. В этом случае экспортируемый из модуля объект становится идентификатором по умолчанию, доступным внутри объекта импортирующего модуля через свойство `default`. Создадим в каталоге `C:\book\p2` файл `test.cjs` со следующим содержанием:

```
function print(msg) {
  console.log(msg);
}
module.exports.print = print;
```

Теперь подключим этот модуль тремя способами внутри файла `app.mjs`:

```
import * as obj from './test.cjs';
obj.default.print('Вариант 1'); // Вариант 1
```

```
import { default as m } from './test.cjs';
m.print('Вариант 2'); // Вариант 2
import test from './test.cjs';
test.print('Вариант 3'); // Вариант 3
```

Первые два варианта приведены лишь для демонстрации доступа к свойству `default`. На практике удобнее третий вариант.

Можно также получить ссылку на функцию `require()` с помощью метода `createRequire()` из модуля `module` и выполнять подключение как обычно:

```
import { createRequire } from 'module';
const require = createRequire(import.meta.url);
const test = require('./test.cjs');
test.print('Вариант 4'); // Вариант 4
```

12.2.12. Подключение модуля ECMAScript внутри модуля CommonJS

Чтобы подключить модуль ECMAScript внутри модуля CommonJS, следует воспользоваться динамическим импортом (см. *разд. 12.2.10*) с помощью инструкции `import()`. Изменим содержимое файла `module1.mjs`, расположенного в каталоге `C:\book\p2`, следующим образом:

```
export let n = 5;
export default class {
  constructor(m) {
    this.model = m;
  }
}
```

Создадим в каталоге `C:\book\p2` файл `test.js` и внутри него подключим файл `module1.mjs`:

```
(async function() {
  let m = {};
  try {
    m = await import('./module1.mjs');
  } catch (e) {
    console.log('Не удалось загрузить модуль.', e);
  }
  console.log(m);
})();
```

Запустим программу в командной строке:

```
C:\book\p2>node test.js
[Module: null prototype] { default: [class default], n: 5 }
```

Как видно из результата, модуль успешно загружен.

12.2.13. Подключение модулей в формате JSON

Создадим в каталоге C:\book\p2 файл myJSON.json со следующим содержимым:

```
{"id":10,"status":true}
```

Теперь попробуем подключить его с помощью инструкции `import`:

```
import * as obj from './myJSON.json';  
console.log(obj);
```

В итоге в Node.js версии 16.0.0 получим исключение:

```
TypeError [ERR_UNKNOWN_FILE_EXTENSION]: Unknown file extension ".json" ❷  
for C:\book\p2\myJSON.json
```

Модули в формате JSON поддерживаются только при явном указании в командной строке флага `--experimental-json-modules`:

```
C:\book\p2>node --experimental-json-modules app.mjs  
(node:14696) ExperimentalWarning: Importing JSON modules is an ❷  
experimental feature. This feature could change at any time  
(Use `node --trace-warnings ...` to show where the warning was created)  
[Module] { default: { id: 10, status: true } }
```

При попытке подключить JSON-модуль Node.js выводит предупреждение о том, что это экспериментальный инструмент и его функциональность может измениться в будущем.

Если загружается файл в формате JSON, то он обрабатывается и возвращается в виде объекта, с которым мы можем работать как обычно. Этот объект доступен как идентификатор по умолчанию, поэтому мы можем написать следующим образом:

```
import json from './myJSON.json';  
console.log(json);           // { id: 10, status: true }  
console.log(json.id);        // 10  
console.log(json.status);    // true
```

Можно также получить ссылку на функцию `require()` с помощью метода `createRequire()` из модуля `module` и выполнять подключение JSON-модуля как обычно, без указания дополнительных флагов:

```
import { createRequire } from 'module';  
const require = createRequire(import.meta.url);  
const json = require('./myJSON.json');  
console.log(json);           // { id: 10, status: true }  
console.log(json.id);        // 10  
console.log(json.status);    // true
```

12.3. Свойства *main*, *exports* и *imports* в файле *package.json*

Свойство `main` внутри файла `package.json` задает точку входа в приложение или пакет. Можно указать только имя запускаемого файла или относительный путь к нему внутри пакета. При запуске программы с указанием имени пакета значение свойства `main` считывается из файла `package.json` и заданный в нем файл запускается. Если пакет подключается, этот файл импортируется. По умолчанию при создании файла `package.json` с помощью NPM в свойстве `main` указано значение `"index.js"`:

```
"main": "index.js"
```

Внутри файла `package.json` в каталоге `C:\book\p2` для свойства `main` у нас задано значение `"app.mjs"`:

```
"main": "app.mjs"
```

Мы можем указать относительный путь к точке входа:

```
"main": "./app.mjs"
```

Изменим содержимое файла `app.mjs` следующим образом:

```
console.log(import.meta.url);
```

Теперь запустим приложение из командной строки, указав имя файла с расширением:

```
C:\book\p2>node app.mjs  
file:///C:/book/p2/app.mjs
```

Если используется модуль CommonJS, расширение `js` можно опустить, а если модуль ECMAScript, то расширение в любом случае нужно указывать, в противном случае возникнет ошибка:

```
C:\book\p2>node app  
Error: Cannot find module 'C:\book\p2\app'
```

Чтобы запустить приложение без указания расширения, следует в командной строке указать флаг `--experimental-specifier-resolution` со значением `node`:

```
C:\book\p2>node --experimental-specifier-resolution=node app  
file:///C:/book/p2/app.mjs
```

Имя приложения можно не указывать — достаточно поставить точку (в результате имя приложения будет считано из файла `package.json`):

```
C:\book\p2>node .  
file:///C:/book/p2/app.mjs
```

Теперь рассмотрим импорт модуля по умолчанию из пакета. Создадим в каталоге `C:\book\p2\node_modules\myPackage` файл `package.json` со следующим содержимым:

```
{
  "name": "myPackage",
  "version": "1.0.0",
  "main": "./main.mjs",
  "type": "commonjs"
}
```

Теперь создадим файл `main.mjs` и добавим в него такой код:

```
console.log(import.meta.url);
```

Подключим наш пакет `myPackage` внутри файла `app.mjs`, указав только его имя:

```
import * as obj from 'myPackage';
```

Запустим на выполнение файл `app.mjs`:

```
C:\book\p2>node app.mjs
file:///C:/book/p2/node_modules/myPackage/main.mjs
```

Также можно загрузить конкретный модуль из пакета, указав путь к нему внутри пакета и его имя с расширением:

```
import * as obj from 'myPackage/main.mjs';
```

Снова запустим на выполнение файл `app.mjs`:

```
C:\book\p2>node app.mjs
file:///C:/book/p2/node_modules/myPackage/main.mjs
```

В последних версиях Node.js точку входа в пакет можно указать также с помощью свойства `exports`, причем при подключении пакета значение свойства `main` будет проигнорировано. Изменим содержимое файла `C:\book\p2\node_modules\myPackage\package.json` следующим образом:

```
{
  "name": "myPackage",
  "version": "1.0.0",
  "main": "./main.mjs",
  "exports": "./module1.mjs",
  "type": "commonjs"
}
```

Теперь добавим в пакет два файла: `module1.mjs` и `module2.mjs` со следующим одинаковым содержимым:

```
console.log(import.meta.url);
```

Подключим наш пакет `myPackage` внутри файла `app.mjs`, указав только его имя:

```
import * as obj from 'myPackage';
```

Запустим на выполнение исправленный файл `app.mjs`:

```
C:\book\p2>node app.mjs
file:///C:/book/p2/node_modules/myPackage/module1.mjs
```

Как видно из результата, вместо файла `main.mjs` был подключен файл `module1.mjs`, указанный в качестве значения свойства `exports`.

Теперь попробуем загрузить конкретный модуль из пакета, указав путь к нему внутри пакета и его имя с расширением:

```
import * as obj1 from 'myPackage/main.mjs';
import * as obj2 from 'myPackage/module1.mjs';
import * as obj3 from 'myPackage/module2.mjs';
```

После запуска файла `app.mjs` на выполнение получим исключение:

```
Error [ERR_PACKAGE_PATH_NOT_EXPORTED]: Package subpath './main.mjs' is
not defined by "exports" in
C:\book\p2\node_modules\myPackage\package.json imported from
C:\book\p2\app.mjs
```

Заметьте, что в предыдущий раз файл `main.mjs` был успешно подключен. А в этот раз попытка импорта любого именованного модуля из пакета приведет к исключению. Дело в том, что свойство `exports` не только задает точку входа, но и ограничивает набор модулей пакета, которые могут быть подключены в других модулях. Если модуль не указан в свойстве `exports`, то подключить его нельзя. Запишем в файл `package.json` код из листинга 12.6.

Листинг 12.6. Указание нескольких модулей в свойстве `exports`

```
{
  "name": "myPackage",
  "version": "1.0.0",
  "main": "./main.mjs",
  "exports": {
    ".": "./module1.mjs",
    "./main.mjs": "./main.mjs",
    "./module1.mjs": "./module1.mjs",
    "./module2": "./module2.mjs",
    "./package.json": "./package.json"
  },
  "type": "commonjs"
}
```

В этот раз в качестве значения свойства `exports` мы указали объект, а не строку. Имена свойств этого объекта задают синтаксис указания пути при импорте. Имя свойства чаще всего будет совпадать с его значением:

```
"./main.mjs": "./main.mjs"
```

Точка входа в пакет задается с помощью свойства с именем в виде точки, например:

```
".": "./module1.mjs"
```

Изменим содержимое файла `app.mjs`:

```
import * as obj from 'myPackage';
```

Теперь запустим программу на выполнение:

```
C:\book\p2>node app.mjs
```

```
file:///C:/book/p2/node_modules/myPackage/module1.mjs
```

Кстати, то, что мы задали модуль `module1.mjs` в качестве точки входа, отнюдь не означает, что мы можем подключить его, указав его имя. Чтобы последнее стало возможным, нужно продублировать разрешение, вот так:

```
"./module1.mjs": "./module1.mjs"
```

Про файл `package.json` также не стоит забывать:

```
"./package.json": "./package.json"
```

Имя свойства может не совпадать с реальным именем модуля. Например, допускается не указывать расширение:

```
"./module2": "./module2.mjs"
```

Изменим содержимое файла `app.mjs`:

```
import * as obj1 from 'myPackage/main.mjs';  
import * as obj2 from 'myPackage/module1.mjs';  
import * as obj3 from 'myPackage/module2';
```

Обратите внимание на последнюю инструкцию. Имя модуля указано без расширения. Если расширение добавить, будет сгенерировано исключение:

```
Error [ERR_PACKAGE_PATH_NOT_EXPORTED]: Package subpath './module2.mjs'   
is not defined by "exports" in   
C:\book\p2\node_modules\myPackage\package.json imported from   
C:\book\p2\app.mjs
```

Теперь запустим программу на выполнение:

```
C:\book\p2>node app.mjs
```

```
file:///C:/book/p2/node_modules/myPackage/main.mjs
```

```
file:///C:/book/p2/node_modules/myPackage/module1.mjs
```

```
file:///C:/book/p2/node_modules/myPackage/module2.mjs
```

Все модули были успешно загружены.

В новых версиях Node.js в составе пути можно использовать символ подстановки `*`, который позволяет разрешить импорт сразу нескольких модулей, подходящих под шаблон. Например, можно разрешить импорт всех модулей из указанного каталога. Пример разрешения импорта всех модулей из каталога `special` (а также точки входа):

```
"exports": {  
  ".": "./module1.mjs",  
  "./special/*": "./special/*.mjs"  
}
```


Пример разрешения импорта всех модулей из пакета:

```
"exports": {
  ".": "./module1.mjs",
  "./": "./"
}
```

Наконец, ничто не мешает указать у свойства произвольное имя, в том числе не совпадающее с именем соответствующего файла:

```
"/mod.mjs": "./module1.mjs"
"/mod2": "./module2.mjs",
```

В этом случае подключить модуль можно, указав имя, совпадающее с именем соответствующего свойства:

```
import * as obj2 from 'myPackage/mod.mjs';
import * as obj3 from 'myPackage/mod2';
```

Объект, указанный в качестве значения свойства `exports`, может содержать следующие свойства (обратите внимание на то, что порядок их следования внутри объекта имеет значение):

- `import` — задает модуль ECMAScript, подключаемый при использовании инструкции `import`;
- `require` — задает модуль CommonJS, подключаемый с помощью функции `require()`;
- `node` — задает подключаемый модуль при запуске программы в Node.js. Значение игнорируется при указании свойств `import` и `require`;
- `default` — модуль, подключаемый по умолчанию. Значение свойства игнорируется при указании свойств `import`, `require` и `node`.

Запишем в файл `C:\book\p2\node_modules\myPackage\package.json` код из листинга 12.7.

Листинг 12.7. Указание объекта в качестве значения свойства `exports` 1

```
{
  "name": "myPackage",
  "version": "1.0.0",
  "main": "./main.mjs",
  "exports": {
    "import": "./main.mjs",
    "require": "./main.cjs",
    "node": "./main.mjs",
    "default": "./main.mjs"
  },
  "type": "commonjs"
}
```

Добавим в каталог `C:\book\p2\node_modules\myPackage` файл `main.cjs` с таким содержанием:

```
console.log(__filename);
```

Содержимое файла `main.mjs` внутри пакета:

```
console.log(import.meta.url);
```

Внутри каталога `C:\book\p2` создадим файл `test.js`:

```
const obj = require('myPackage');
```

Исправим содержимое файла `app.mjs` на следующее:

```
import * as obj from 'myPackage';
```

Если мы запустим на выполнение файл `app.mjs`, будет импортирован модуль, указанный в свойстве `import`:

```
C:\book\p2>node app.mjs
```

```
file:///C:/book/p2/node_modules/myPackage/main.mjs
```

Если свойство `import` отсутствует, будет использовано значение свойства `node`. При отсутствии свойства `node` будет задействовано значение свойства `default`. Поэтому порядок следования свойств внутри объекта имеет значение.

Если же мы запустим файл `test.js`, то содержимое свойства `import` будет проигнорировано. В результате будет загружен модуль, путь к которому указан с помощью свойства `require`. Пример:

```
C:\book\p2>node test.js
```

```
C:\book\p2\node_modules\myPackage\main.cjs
```

В качестве значения свойства `node` можно указать объект со свойствами `import` и `require`. Запишем в файл `package.json` код из листинга 12.8.

Листинг 12.8. Указание объекта в качестве значения свойства `exports` 2

```
{
  "name": "myPackage",
  "version": "1.0.0",
  "main": "./main.mjs",
  "exports": {
    "node": {
      "import": "./main.mjs",
      "require": "./main.cjs"
    },
    "default": "./main.mjs"
  },
  "type": "commonjs"
}
```

Запустим программы на выполнение:

```
C:\book\p2>node app.mjs
file:///C:/book/p2/node_modules/myPackage/main.mjs
```

```
C:\book\p2>node test.js
C:\book\p2\node_modules\myPackage\main.cjs
```

Таким образом мы можем указать точку входа внутри пакета, содержащего одновременно два типа модулей. Напомним, что файлы с расширением `mjs` всегда обрабатываются как модули ECMAScript, а файлы с расширением `cjs` — как модули CommonJS. Файлы с расширением `js` обрабатываются в зависимости от значения свойства `type` внутри файла `package.json`. Если свойство `type` имеет значение `"commonjs"` (или вообще не задано), то файлы с расширением `js` обрабатываются как модули CommonJS, а если свойство имеет значение `"module"`, — то как модули ECMAScript.

В предыдущих примерах мы описали универсальную точку входа в пакет. Однако теперь не можем подключить другие модули из пакета, поскольку не пометили их как доступные для подключения. Чтобы решить эту проблему, объект со свойствами `import`, `require`, `node` и `default` нужно присвоить свойству, описывающему путь. Запишем в файл `package.json` код из листинга 12.9.

Листинг 12.9. Указание в объекте из свойства `exports` свойств, описывающих пути к модулям

```
{
  "name": "myPackage",
  "version": "1.0.0",
  "main": "./main.mjs",
  "exports": {
    ".": {
      "import": "./main.mjs",
      "require": "./main.cjs",
      "node": "./main.mjs",
      "default": "./main.mjs"
    },
    "./main": {
      "node": {
        "import": "./main.mjs",
        "require": "./main.cjs"
      },
      "default": "./main.mjs"
    },
    "./module1.mjs": "./module1.mjs",
    "./module2": "./module2.mjs",
    "./package.json": "./package.json"
  },
  "type": "commonjs"
}
```

Отредактируем файл `app.mjs`:

```
import * as obj1 from 'myPackage';
import * as obj2 from 'myPackage/main';
import * as obj3 from 'myPackage/module1.mjs';
import * as obj4 from 'myPackage/module2';
console.log( obj1 == obj2 );
```

Запустим программу на выполнение:

```
C:\book\p2>node app.mjs
file:///C:/book/p2/node_modules/myPackage/main.mjs
file:///C:/book/p2/node_modules/myPackage/module1.mjs
file:///C:/book/p2/node_modules/myPackage/module2.mjs
true
```

Отредактируем файл `test.js`:

```
const obj1 = require('myPackage');
const obj2 = require('myPackage/main');
console.log( obj1 == obj2 );
import('myPackage/module1.mjs');
import('myPackage/module2');
```

Запустим программу на выполнение:

```
C:\book\p2>node test.js
C:\book\p2\node_modules\myPackage\main.cjs
true
file:///C:/book/p2/node_modules/myPackage/module1.mjs
file:///C:/book/p2/node_modules/myPackage/module2.mjs
```

Теперь мы имеем две точки входа в зависимости от типа импорта, а также нам доступны для подключения другие модули пакета. В результатах мы получили на одну инструкцию вывода меньше, т. к. модули кэшируются. Чтобы показать, что объекты модулей одинаковые, мы выводим в консоль результат их сравнения. В обоих случаях мы получили значение `true`.

Может случиться так, что имеются два модуля, выполняющие одинаковую задачу, но относящиеся к разным типам — ECMAScript и CommonJS — и, возможно, имеющие разные имена (назовем их *подставляемыми*, почему — будет сказано далее). И требуется подключить к другому модулю один из этих модулей, относящийся к тому же типу, что и подключающий модуль. Вместо того чтобы вспоминать, какое имя имеет модуль подходящего типа, можно указать у обоих этих модулей *псевдоним*, *правила подстановки* и в дальнейшем выполнять подключение по заданному псевдониму.

Сначала подставляемым модулям следует, собственно, дать псевдоним, аналогичный имени переменной, только с предшествующим символом решетки (`#`). Далее в файле `package.json` необходимо описать правила подстановки модулей. Они записываются в свойстве `imports` в виде объекта, имя каждого из свойств которого должно совпадать с выбранным ранее псевдонимом подставляемых модулей, а зна-

чением должен стать вложенный объект. Имя каждого из свойств вложенного объекта должно совпадать с обозначением типа импорта (`import`, `require`, `node` или `default`, их описание было приведено ранее), а значение свойства укажет имя подставляемого модуля, импортируемого в случае применения соответствующего типа импорта.

В качестве примера создадим в каталоге `C:\book\p2\node_modules\myPackage` модули `print.mjs` (стандарта ECMAScript) и `pr.cjs` (CommonJS). Содержимое модуля `print.mjs`:

```
console.log(import.meta.url);
```

Содержимое модуля `pr.cjs`:

```
console.log(__filename);
```

Придумаем для обоих модулей псевдоним `#print`.

Запишем в файл `package.json` код из листинга 12.10.

Листинг 12.10. Правила подстановки модулей

```
{
  "name": "myPackage",
  "version": "1.0.0",
  "main": "./main.mjs",
  "exports": {
    "import": "./main.mjs",
    "require": "./main.cjs"
  },
  "imports": {
    "#print": {
      "import": "./print.mjs",
      "require": "./pr.cjs"
    }
  },
  "type": "commonjs"
}
```

Теперь при подключении псевдонима `#print` внутри модуля ECMAScript будет подключен модуль `print.mjs`, а при подключении внутри модуля CommonJS — подключаемый модуль `pr.cjs`.

Изменим содержимое файла `main.mjs` внутри пакета на следующее:

```
import * as obj from '#print'
```

а содержимое файла `main.cjs` — на следующее:

```
const obj = require('#print');
```

Не забываем, что при подключении псевдоним подставляемых пакетов также указывается с добавленным впереди символом `#`.

Запустим на исполнение файл `app.mjs`:

```
C:\book\p2> node app.mjs
file:///C:/book/p2/node_modules/myPackage/print.mjs
```

а потом — файл `test.js`:

```
C:\book\p2> node test.js
C:\book\p2\node_modules\myPackage\pr.cjs
```

Как видим, подстановка модулей работает.

12.4. Переменные уровня приложения

Как вы уже знаете, глобальные переменные, объявленные в модуле, на самом деле видны лишь внутри текущего модуля, но недоступны для других модулей. В веб-браузере глобальные переменные, объявленные внутри простого файла (не модуля), автоматически становятся свойствами глобального объекта `window`.

В Node.js существует аналогичный глобальный объект `global`. Все глобальные функции и объекты, которые мы использовали ранее, доступны как свойства этого объекта. Например, эти инструкции эквиваленты:

```
console.log('test');           // test
global.console.log('test');    // test
```

Если нужно, чтобы переменная стала глобальной для всего приложения (*переменная уровня приложения*), достаточно сделать ее свойством объекта `global`. Изменим содержимое файла `C:\book\p2\module1.mjs` следующим образом:

```
global.myVar2 = 10;
export function test() {
  console.log(global.myVar1);
}
```

Здесь мы объявляем переменную уровня приложения `myVar2`, а также экспортируем функцию. Внутри функции выводим значение переменной уровня приложения `myVar1`, которую объявим в основном файле. Содержимое файла `app.mjs`:

```
import { test } from './module1.mjs';
global.myVar1 = 5;
test();           // 5
console.log(global.myVar2); // 10
console.log(myVar1);    // 5
console.log(myVar2);    // 10
```

Вначале подключаем модуль и импортируем функцию `test()`. Далее объявляем переменную уровня приложения `myVar1`. Вызываем функцию `test()` и получаем в консоли значение переменной `myVar1`. Точно так же мы можем обратиться к переменной `myVar2`, объявленной внутри модуля `module1.mjs`.

ОБРАТИТЕ ВНИМАНИЕ

При объявлении переменной обязательно нужно указывать объект `global`, а вот при получении значения объект `global` можно не указывать.

Область видимости идентификаторов внутри модуля ограничена не просто так. Благодаря этому можно избежать конфликта имен. Поэтому старайтесь не использовать переменные уровня приложения без особой необходимости. В нашем случае переменную `myVar1` можно передать в качестве параметра функции `test()`, а переменную `myVar2` — просто экспортировать из модуля. Изменим код модуля `module1.mjs`:

```
export let myVar2 = 10;
export function test(n) {
  console.log(n);
}
```

Содержимое файла `app.mjs`:

```
import { test, myVar2 } from "./module1.mjs";
let myVar1 = 5;
test(myVar1);                // 5
console.log(myVar2);         // 10
```

Как видно из результата, удалось обойтись без переменных уровня приложения. Однако мы не можем поменять значение `myVar2`, ведь после импорта идентификатор становится константой. Ну, нам ведь это и не нужно. А если станет нужно, мы просто сделаем значение объектом.

Идентификатор `globalThis` хранит ссылку на текущий глобальный объект: `window`, если код выполняется в веб-браузере, и `global` — в случае Node.js. Так что в приведенных ранее примерах мы можем поменять идентификатор `global` на `globalThis`:

```
// Модуль C:\book\p2\module1.mjs
globalThis.myVar2 = 10;
export function test() {
  console.log(globalThis.myVar1);
}

// Файл C:\book\p2\app.mjs
import { test } from "./module1.mjs";
globalThis.myVar1 = 5;
test();                // 5
console.log(globalThis.myVar2); // 10
console.log(myVar1);    // 5
console.log(myVar2);    // 10
```

и получить тот же результат. Идентификатор `globalThis` удобно использовать в коде, который должен работать и в веб-браузерах, и в Node.js.



ГЛАВА 13

Обработка ошибок

От ошибок никто не застрахован, поэтому очень важно знать, как быстро найти ошибку.

ПРИМЕЧАНИЕ

В последующих примерах мы возвращаемся в проект `p1` и будем запускать на выполнение файл `index.js` из каталога `C:\book\p1`. Не забудьте сменить каталог в командной строке, если запускали программы из предыдущей главы.

13.1. Типы ошибок

Бывают три вида программных ошибок: синтаксические, логические и ошибки времени выполнения.

- ❑ *Синтаксические ошибки* — это ошибки в синтаксисе языка (неправильное написание оператора, отсутствие закрывающей или открывающей скобки и т. п.). Имеют тип `SyntaxError`. Как правило, интерпретатор предупредит о наличии такой ошибки, а программа не будет выполняться совсем. Например, если вместо:

```
console.log('Строка');
```

написать (нет закрывающей кавычки):

```
console.log('Строка);
```

то в консоли отобразятся следующие сообщения:

```
C:\book\p1>node index.js
```

```
C:\book\p1\index.js:1
```

```
console.log('Строка);  
^^^^^^^^
```

```
SyntaxError: Invalid or unexpected token  
    at wrapSafe (internal/modules/cjs/loader.js:1053:16)  
    at Module._compile (internal/modules/cjs/loader.js:1101:27)  
    at Object.Module._extensions..js  
      (internal/modules/cjs/loader.js:1157:10)
```



```

at Module.load (internal/modules/cjs/loader.js:985:32)
at Function.Module._load
(internal/modules/cjs/loader.js:878:14)
at Function.executeUserEntryPoint [as runMain]
(internal/modules/run_main.js:71:12)
at internal/main/run_main_module.js:17:47

```

Примерное место ошибки показывают символы `^`. Номер строки с ошибкой (1) и имя файла (`C:\book\p1\index.js`) отображаются во второй строке. Достаточно найти строку с указанным номером в исходном коде, добавить закрывающую кавычку и заново запустить программу на выполнение.

Редактор VS Code подсвечивает точку с запятой красным цветом еще до запуска программы. Если навести указатель мыши на точку с запятой, отобразится всплывающее окно с сообщением:

Строковый литерал без признака конца. `ts(1002)`

Обращайте внимание на различные выделения цветом и подчеркивания в редакторе. Так вы обнаружите ошибку еще до запуска программы.

Перечислим наиболее часто встречающиеся синтаксические ошибки:

- опечатка в имени оператора или функции;
- буква набрана в русской раскладке клавиатуры вместо латинской;
- неправильный регистр букв;
- отсутствие открывающей или закрывающей скобки (или наоборот, лишние скобки);
- в цикле `for` параметры указаны через запятую, а не через точку с запятой.

□ *Логические ошибки* — это ошибки в логике работы программы, которые можно выявить только по результатам ее выполнения. Как правило, интерпретатор не предупреждает о наличии ошибки, и программа будет выполняться, поскольку не содержит синтаксических ошибок. Логические ошибки достаточно трудно выявить.

Предположим, необходимо вывести первые три элемента массива. Программист, забыв, что нумерация массивов начинается с 0, пишет следующий код:

```

let arr = [1, 2, 3, 4];
for (let i = 1; i < 4; i++) console.log(arr[i]);

```

В итоге возникает логическая ошибка, т. к. будут получены не первые элементы массива, а три элемента, начиная со второго. Так как в данном примере нет синтаксических ошибок, интерпретатор сочтет код правильным.

Если в логическом выражении вместо оператора `==` (равно) указан оператор присваивания `=`, это также приведет к логической ошибке:

```

let x = 5;
if (x = 6) console.log('Переменная x равна 6');
else console.log('Переменная x не равна 6');

```

Этот код выведет совсем не то, что хотел программист:

Переменная `x` равна 6

- ❑ **Ошибки времени выполнения** — это ошибки, которые возникают во время работы программы. Могут быть как следствием логических ошибок, так и вызываться внешними событиями (например, отсутствием нужного файла, отсутствием прав на запись в файл и др.). В этом случае интерпретатор генерирует исключение. Если в коде не предусмотрена обработка исключения, программа прерывается и выводится сообщение об ошибке.

В некоторых языках (например, в PHP) ошибки времени выполнения возникают из-за деления на ноль или обращения к несуществующему элементу массива. В JavaScript в этих случаях программа прервана не будет. При попытке деления на ноль возвращается значение `Infinity` или `NaN`. Пример:

```
console.log(5 / 0); // Infinity
console.log(0 / 0); // NaN
```

При обращении к несуществующему элементу массива возвращается значение `undefined`:

```
let arr = [ 1, 2 ];
console.log(arr[20]); // undefined
```

Очень часто ошибки времени выполнения возникают при использовании условий:

```
let x = 6;
if (x > 5) console.log('x > 5');
else conole.log(x); // Строка с ошибкой (conole вместо console)
```

В этом примере никакой ошибки не будет, пока соблюдается условие `x > 5`. Как только условие перестанет выполняться, сразу возникнет ошибка, и выполнение программы будет прервано:

```
ReferenceError: conole is not defined
```

13.2. Исключения и их обработка.

Инструкция ***try...catch...finally***

Исключение — это объект, создаваемый самим интерпретатором при возникновении ошибки, служащий сигналом о возникновении ошибки и хранящий исчерпывающие сведения о возникшей ошибке. Программа может осуществлять *обработку исключений* — выполнение указанных действий при возникновении исключения, при этом исполнение программы не прерывается.

Для обработки исключений предназначена инструкция `try...catch`. Формат инструкции:

```
try {
    <Блок, в котором может возникнуть исключение>
}
```

```
[catch ([<Ссылка на объект Error>]) {
  <Блок, выполняемый при возникновении исключения>
}]
[finally {
  <Блок, выполняемый в любом случае>
}]
```

Инструкции, в которых могут возникнуть исключения, располагаются внутри блока `try`. Если при выполнении этих инструкций возникнет исключение, управление будет передано в блок `catch`. В качестве параметра в блоке `catch` можно указать переменную, через которую будет доступен объект возникшего исключения. Если же исключение не возникнет, инструкции внутри блока `catch` не выполнятся.

Инструкции внутри блока `finally` будут выполнены независимо от того, возникло в блоке `try` исключение или нет. Обычно внутри блока `finally` выполняется освобождение каких-либо ресурсов (например, закрытие ранее открытых файлов). Блоки `catch` и `finally` являются необязательными, но хотя бы один из них должен быть указан. Пример обработки исключения:

```
let x = 3;
try {
  if (x > 5) console.log('x > 5');
  else console.log(x); // Строка с ошибкой
} catch (e) {
  console.error(`Возникла ошибка: ${e.name}: ${e.message}`);
} finally {
  console.log('Сообщение будет выведено в любом случае');
}
```

Результат в консоли при ошибке:

```
Возникла ошибка: ReferenceError: console is not defined
Сообщение будет выведено в любом случае
```

Вывести сообщение об ошибке в консоль позволяет метод `error()` объекта `console`. Метод имеет следующие форматы:

```
console.error(<Данные 1>[, . . . , <Данные N>])
console.error(<Строка формата>, <Данные 1>[, . . . , <Данные N>])
```

Инструкции `try...catch` можно вкладывать друг в друга. Если внутри вложенного обработчика нет блока `catch`, или внутри блока `catch` генерируется исключение, то исключение "всплывает" к обработчику более высокого уровня. Пример вложения обработчиков:

```
try {
  try {
    console.log(''); // Строка с ошибкой
    console.log('Это сообщение не выводится');
  } finally {
    console.log('Сообщение будет выведено в любом случае');
  }
}
```

```
    console.log('Это сообщение то же не выводится');  
  } catch (e) {  
    console.error(`Возникла ошибка: ${e.name}: ${e.message}`);  
  }  
  console.log('Инструкция после обработчика');
```

Результат:

Сообщение будет выведено в любом случае
Возникла ошибка: ReferenceError: conole is not defined
Инструкция после обработчика

При возникновении исключения внутри вложенного блока `try` управление сначала передается в блок `finally`, а затем — сразу в блок `catch` внешнего обработчика. Код, расположенный после инструкции, которая сгенерировала исключение, выполнен не будет, поэтому текст "Это сообщение не выводится" никогда выведен не будет. После выполнения инструкций в блоке `catch` управление передается инструкции, расположенной сразу после внешнего `try...catch`, поскольку интерпретатор считает, что исключение обработано и можно продолжить выполнение программы.

Иногда необходимо не продолжить выполнение программы после исключения, а остановить ее. В этом случае внутри блока `catch` можно произвести завершающие действия (например, проинформировать пользователя о проблеме), а затем прервать работу программы с помощью метода `exit()` из модуля `process` (см. *разд. 2.11*):

```
try {  
  conole.log(''); // Строка с ошибкой  
} catch (e) {  
  console.error(`Возникла ошибка: ${e.message}`);  
  process.exit(1); // Прерываем выполнение программы  
}  
console.log('Это сообщение не выводится');
```

Результат:

Возникла ошибка: conole is not defined

Если исключение нигде не обрабатывается, управление передается обработчику по умолчанию, который останавливает выполнение программы и выводит стандартную информацию об ошибке.

Для быстрой вставки инструкции `try...catch` в редакторе VS Code предусмотрена следующая комбинация:

```
trycatch =>  
try {  
  
} catch (error) {  
  
}
```

13.3. Класс *Error*: объекты исключения

В качестве параметра в блоке `catch` можно указать переменную, через которую будет доступен объект исключения, относящийся к классу `Error` или к одному из его подклассов. С помощью оператора `instanceof` можно проверить класс исключения и для разных классов выполнить различные инструкции:

```
try {
  console.log(''); // Строка с ошибкой
} catch (e) {
  if (e instanceof ReferenceError) {
    console.log('Ошибка ReferenceError');
  }
  else {
    console.log(`Другая ошибка: ${e.name}: ${e.message}`);
  }
} // Результат: Ошибка ReferenceError
```

Конструктор класса `Error`:

```
<Объект> = new Error(<Описание ошибки>);
```

Пример создания объекта и вывода информации об исключении:

```
let err = new Error('Описание ошибки');
console.log(err.name); // Error
console.log(err.message); // Описание ошибки
console.log(err.stack);
/*
Error: Описание ошибки
    at Object.<anonymous> (C:\book\pl\index.js:1:11)
    at Module._compile (internal/modules/cjs/loader.js:1137:30)
    at Object.Module._extensions..js
    (internal/modules/cjs/loader.js:1157:10)
    at Module.load (internal/modules/cjs/loader.js:985:32)
    at Function.Module._load
    (internal/modules/cjs/loader.js:878:14)
    at Function.executeUserEntryPoint [as runMain]
    (internal/modules/run_main.js:71:12)
    at internal/main/run_main_module.js:17:47
*/
```

Получить информацию об исключении позволяют следующие основные свойства класса `Error`:

- ☐ `name` — имя класса исключения в виде строки;
- ☐ `message` — описание ошибки, заданное при создании объекта;
- ☐ `stack` — стек вызовов.

Существует несколько стандартных классов исключений, наследующих класс `Error`: `EvalError`, `SyntaxError`, `RangeError`, `ReferenceError`, `TypeError`, `URIError` и др. Можно создать свой собственный класс исключения, например:

```
class MyError extends Error {
  constructor(msg, n) {
    super(msg);
    this.name = 'MyError';
    this.n = n;
  }
}

let err = new MyError('Описание ошибки', 10);
console.log(err.name);    // MyError
console.log(err.message); // Описание ошибки
console.log(err.n);       // 10
```

13.4. Оператор *throw*: генерирование исключений

В некоторых случаях может потребоваться принудительно сгенерировать исключение. Например, какая-либо функция, принимающая параметр строго определенного типа, может генерировать исключение, чтобы указать разработчику, что он передал функции параметр недопустимого типа.

Для генерирования исключения применяется оператор `throw <Объект исключения>`:

```
let x = -3;
try {
  if (x < 0)
    throw new Error('Переменная не может быть меньше нуля');
} catch (e) {
  console.error(`${e.name}: ${e.message}`);
}
```

Пример генерирования исключения в блоке `catch` обработчика исключений с целью передачи этого исключения вышестоящему обработчику:

```
let x = -3;
try {
  try {
    if (x < 0)
      throw new Error('Переменная не может быть меньше нуля');
  } catch (e) {
    throw e; // Повторная генерация исключения
  }
} catch (e) {
  console.error(`${e.name}: ${e.message}`);
}
```

Вместо объекта исключения в операторе `throw` можно указать любое другое значение: логическую величину, число, строку, объект другого класса и др. Это значение будет доступно через параметр в блоке `catch`. Пример:

```
try {
  // throw true;
  // throw 10;
  // throw 'Error';
  throw { name: 'MyError', message: 'Описание ошибки' };
} catch (e) {
  console.error(typeof e);
  if (typeof e === 'object')
    console.error(`${e.name}: ${e.message}`);
  else
    console.error(e);
}
```

ПРИМЕЧАНИЕ

Нельзя генерировать исключение внутри асинхронного обработчика ошибок.

13.5. Поиск ошибок в программе

В предыдущих разделах мы научились обрабатывать ошибки времени выполнения. Однако большую часть времени программист затрачивает на поиск и устранение логических ошибок, вызывающих выдачу программой неверного результата. Ситуация еще более осложняется, когда неверный результат проявляется лишь периодически, а не постоянно. Инсценировать такую же ситуацию, чтобы получить этот же неверный результат, бывает крайне сложно, и такой процесс занимает много времени. В этом разделе мы рассмотрим способы поиска ошибок и дадим советы по оформлению кода, что будет способствовать быстрому поиску ошибок.

Первое, на что следует обратить внимание, — это форматирование кода, которое позволит быстрее найти ошибку. Перед всеми инструкциями внутри блока должно быть расположено одинаковое количество пробелов (обычно три или четыре). От применения символов табуляции лучше отказаться. Если все же их используете, то не следует в одном файле совмещать и пробелы, и табуляцию. Для вложенных блоков количество пробелов умножают на уровень вложенности. Если для блока первого уровня вложенности использовалось три пробела, то для блока второго уровня вложенности поставьте шесть пробелов, для третьего уровня — девять пробелов и т. д. Пример форматирования вложенных блоков приведен в листинге 13.1.

Листинг 13.1. Пример форматирования вложенных блоков

```
let arr = [
  [1, 2, 3, 4],
  [5, 6, 7, 8]
];
```

```
let i = 0, j = 0;
for (i = 0; i < arr.length; i++) {
  for (j = 0; j < arr[i].length; j++) {
    process.stdout.write(`${arr[i][j]} `);
  }
  process.stdout.write('\n');
}
```

Открывающая фигурная скобка может быть расположена как на одной строке с оператором, так и на следующей строке. Какой способ выбрать, зависит от предпочтений программиста или от требований по оформлению кода, принятых внутри фирмы. Пример размещения открывающей фигурной скобки на отдельной строке:

```
for (i = 0; i < arr.length; i++)
{
  for (j = 0; j < arr[i].length; j++)
  {
    process.stdout.write(`${arr[i][j]} `);
  }
  process.stdout.write('\n');
}
```

Одна строка кода не должна содержать более 80 символов. Если число символов больше, то следует выполнить переход на новую строку. При этом продолжение смещается относительно основной инструкции на величину отступа или выравнивается по какому-либо элементу. Иначе строка не будет помещаться на стандартном экране и придется пользоваться горизонтальной полосой прокрутки, что неудобно.

Если программа слишком большая, следует задуматься о разделении ее на отдельные функции или классы, которые выполняют логически законченные действия. Прежде чем вставить функцию (или класс) в основную программу, ее нужно протестировать в отдельном проекте, передавая функции различные значения и проверяя результат выполнения.

Имейте в виду, что форматировать код следует при написании программы, а не в процессе поиска ошибок. Этим вы сократите время поиска ошибки и, скорее всего, заметите ошибку еще на этапе написания. Если все же ошибка возникла, то вначале следует инсценировать ситуацию, при которой ошибка проявляется. После этого можно начать поиск ошибки.

НА ЗАМЕТКУ

Редактор VS Code форматирует код автоматически. Описание этой возможности приведено в разд. 13.9.

Причиной периодических ошибок чаще всего являются некорректные внешние данные. Например, если числа вводятся пользователем, а затем выполняется их деление, то вполне возможна ситуация, при которой пользователь введет число 0. Деление на ноль приведет к ошибке. Следовательно, все данные, которые поступа-

ют от пользователей, должны проверяться на соответствие допустимым значениям. Если данные им не соответствуют, то нужно вывести сообщение об ошибке, а затем повторно запросить новое число или прервать выполнение всей программы. Кроме того, нужно учесть, что пользователь может ввести вовсе не число, а строку.

Метод `log()` объекта `console` удобно использовать для вывода промежуточных значений. В этом случае значения переменных сначала выводятся в самом начале программы, и выполняется проверка соответствия полученных значений ожидаемым. Если все в порядке, то инструкция с методом `log()` перемещается на следующую строку программы и опять производится проверка и т. д. Если значения не совпали с ожидаемыми, значит, ошибка возникает в инструкции, расположенной перед инструкцией с методом `log()`. Если ошибка в пользовательской функции, то проверку значений проводят внутри функции, каждый раз перемещая инструкцию с выводом значений.

13.6. Метод `assert()`

Метод `assert()` объекта `console` выводит сообщение в консоль только в том случае, если заданное логическое выражение возвращает значение `false`. Выполнение программы при этом не прерывается. Формат метода:

```
assert([<Логическое выражение>[, ...<Данные>]])
```

Если параметры не заданы, то просто выводится сообщение:

```
console.assert(); // Assertion failed
```

Значение необязательного параметра `<Данные>` будет выведено после сообщения. Также можно во втором параметре указать строку со спецификаторами формата, а в последующих параметрах — подставляемые значения. Пример:

```
let x = -1;
console.assert(x < 0, `${x} >= 0`); // Игнорируется
console.assert(x > 0, `${x} <= 0`); // Assertion failed: -1 <= 0
console.assert(x > 0); // Assertion failed
console.assert(x > 0, '%d %s', x, '<= 0'); // Assertion failed: -1 <= 0
```

Вызовы метода `assert()` можно расставить на этапе написания кода и не удалять после отладки. Если заданное логическое выражение вернет `true`, то никакое сообщение в консоль выводиться не будет.

13.7. Отладка программы в редакторе VS Code

Сделать поиск ошибок более эффективным позволяет отладчик, встроенный в редактор VS Code. С его помощью можно выполнять программу по шагам, на каждом шаге контролируя значения переменных и удостовераясь, что порядок выполнения инструкций соответствует разработанному алгоритму. В качестве примера будем отлаживать программу из листинга 13.2, которая содержит две наиболее часто

встречающиеся ошибки: выход за пределы массива и нумерация элементов массива с 1, а не с 0.

Листинг 13.2. Отладка программы

```
function print(value) {                                // 1
    let s = `${value}`.padStart(3);                    // 2
    process.stdout.write(s);                            // 3
}                                                        // 4
let arr = [                                             // 5
    [1, 2, 3, 4],                                       // 6
    [5, 6, 7, 8]                                       // 7
];                                                       // 8
let i = 0, j = 0;                                       // 9
for (i = 0; i <= arr.length; i++) {                    // 10
    for (j = 1; j < arr[i].length; j++) {               // 11
        print( arr[i][j] );                             // 12
    }                                                    // 13
    process.stdout.write('\n');                         // 14
}                                                        // 15
```

Прежде чем начать отладку, необходимо пометить строки внутри программы с помощью *точек останова*. На точке останова обычное выполнение программы приостановится, и мы сможем далее выполнить программу по шагам.

Для добавления точки останова достаточно щелкнуть слева от нужной строки — в том месте появится красный кружок. Повторный щелчок на точке останова убирает ее. Также можно сделать строку активной, а затем в меню **Выполнить** выбрать пункт **Переключить точку останова** или нажать клавишу <F9>. Повторное действие уберет точку останова. Чтобы временно отключить, включить или удалить точку останова, следует щелкнуть на ней правой кнопкой мыши и в открывшемся контекстном меню выбрать пункт соответственно **Отключить**, **Включить** или **Удалить**. Установим в нашей программе две точки останова: на строках 9 и 12.

Отладка программы выполняется на панели **Запуск и отладка**, вывести которую можно щелчком на соответствующем значке в боковой панели. В нижней части панели расположен раздел **Точки останова**, в котором отображаются все установленные точки останова. С помощью этого раздела можно управлять точками останова: временно отключать их, вновь включать, удалять и т. д. И сразу же выведем консоль, в которой будут выводиться результаты отладки, выбрав в меню **Вид** пункт **Консоль отладки** или нажав комбинацию клавиш <Shift>+<Ctrl>+<Y>.

Чтобы начать отладку, следует в меню **Выполнить** выбрать пункт **Запустить отладку** или нажать клавишу <F5>. Отобразится список (рис. 13.1), в котором нужно выбрать тип отладки. Выбираем пункт **Node.js**. После запуска отладки интерфейс редактора VS Code будет выглядеть так, как показано на рис. 13.2. При достижении точки останова выполнение программы приостановится и отладчик станет ждать

дальнейших действий программиста. Инструкция, которая будет выполняться на следующем шаге, будет помечена желтым фоном. Чтобы прервать отладку, в меню **Выполнить** выбираем пункт **Остановить отладку** или нажимаем комбинацию клавиш <Shift>+<F5>.

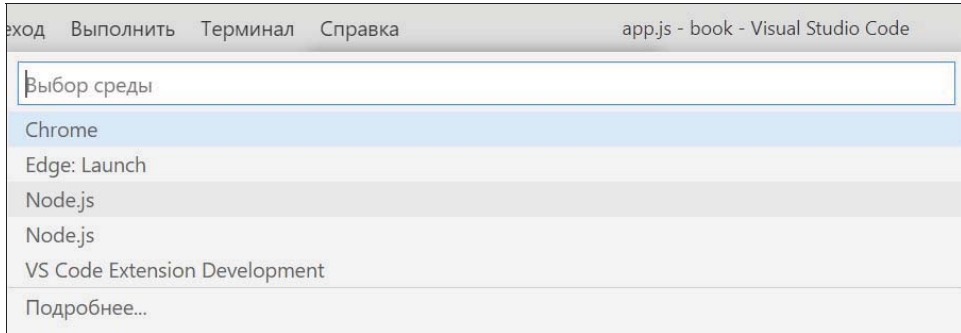


Рис. 13.1. Список для выбора типа отладки

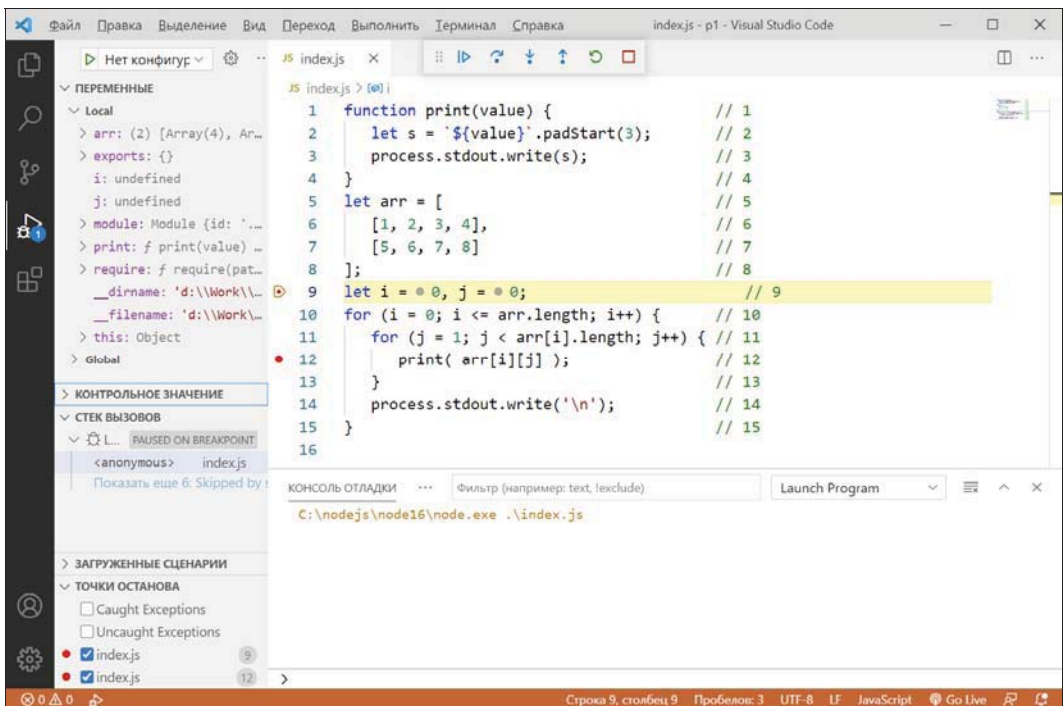


Рис. 13.2. Окно редактора VS Code после запуска отладки

Рассмотренный способ запуска нам не подходит, т. к. мы выводим данные с помощью метода `write()` объекта `stdout`, а не методов объекта `console`. В результате мы не увидим результатов вывода. Кроме того, отладку обычно запускают несколько раз, и каждый раз выбирать тип отладки при запуске быстро надоедает.

Создадим конфигурационный файл отладки и внутри него запишем все необходимые настройки. Остановим отладку. Далее в меню **Выполнить** выберем пункт **Добавить конфигурацию** и из открывшегося списка выберем пункт **Node.js**. В результате в каталоге `C:\book\p1\vscode` будет создан файл `launch.json`, и его содержимое отобразится на отдельной вкладке. Изменим содержимое этого файла, как показано в листинге 13.3.

Листинг 13.3. Файл `launch.json`

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Отладка программы",
      "runtimeExecutable": "node",
      "runtimeArgs": [],
      "args": [],
      "outputCapture": "std",
      "skipFiles": [
        "<node_internals>/**"
      ],
      "program": "${workspaceFolder}\\index.js"
    }
  ]
}
```

Все записанные в этом файле конфигурации (которых может быть произвольное количество) перечисляются в свойстве `configurations` в виде массива. Элементами этого массива являются объекты, описывающие конкретные конфигурации. Рассмотрим основные свойства этих объектов (полный список приведен в документации):

- ❑ `type` — название исполняющей среды, в которой запускается отлаживаемая программа (например, `"node"`);
- ❑ `request` — способ запуска отлаживаемой программы (`"launch"` — обычный запуск);
- ❑ `name` — произвольное имя конфигурации, которое будет отображаться в списке. Может содержать любые символы, в том числе буквы кириллицы и пробелы;
- ❑ `runtimeExecutable` — путь к программе `node.exe`. Свойство полезно при запуске с помощью версии, путь к которой не задан в переменной окружения `PATH`. Значение по умолчанию: `"node"`;
- ❑ `runtimeArgs` — массив с параметрами, передаваемыми программе `node.exe`;
- ❑ `args` — массив с параметрами, передаваемыми нашей программе;

- ❑ `outputCapture` — имя объекта, представляющего поток вывода. По умолчанию имеет значение `"console"` (объект `console`). Чтобы отображались данные, выводимые с помощью методов стандартных потоков вывода, следует указать значение `"std"`;
- ❑ `skipFiles` — массив с путями файлов, которые не будут обрабатываться отладчиком;
- ❑ `program` — полный путь к запускаемому файлу.

Чтобы добавить новую конфигурацию в открытый файл `launch.json`, следует нажать кнопку **Добавить конфигурацию** или комбинацию клавиш `<Ctrl>+<Пробел>`. В результате отобразится список (рис. 13.3) с возможными типами отладки. Выбираем нужный тип и редактируем при необходимости значения свойств, сгенерированные автоматически.

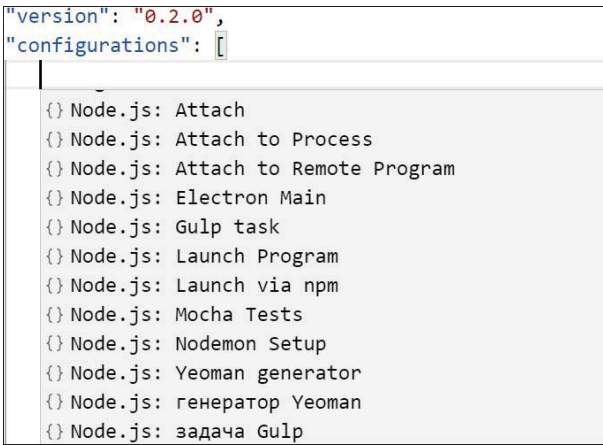


Рис. 13.3. Выбор типа конфигурации

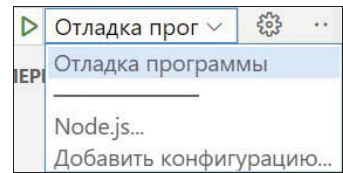


Рис. 13.4. Раскрывающийся список конфигураций отладки на панели **Запуск и отладка**

После создания конфигурационного файла на панели **Запуск и отладка** появится раскрывающийся список, в котором можно выбрать наш тип отладки **Отладка программы** (рис. 13.4). Слева от списка расположен значок с изображением треугольника, с помощью которого можно запустить отладку. Запускаем ее.

Запустить отладку можно и из строки состояния, при этом неважно, какая панель открыта в данный момент. Достаточно найти в левой части строки состояния кнопку со значком в виде стрелки вправо и именем последней использованной конфигурации отладки (например, **Отладка программы**) и щелкнуть ее. На экране появится список всех имеющихся конфигураций отладки, в котором следует выбрать нужную.

После запуска отладчика выполнение программы приостановится на строке 9 — в том месте, где установлена первая точка останова. В режиме прерывания можно посмотреть значение какой-либо переменной, наведя курсор мыши на ее имя. Попробуйте навести указатель на имя массива `arr` — во всплывающем окне отобра-

зятся значения всех элементов массива. Если навести указатель мыши на переменные `i` и `j`, то получим значение `undefined`, т. к. инструкция, создающая эти переменные, еще не выполнялась. Их значение мы увидим только после следующего шага.

Посмотреть значения сразу всех переменных можно в разделе **Переменные** панели **Запуск и отладка** (см. рис. 13.2). Также можно контролировать значения отдельных переменных, предварительно занеся их в раздел **Контрольное значение** (рис. 13.5). Сделать это можно, щелкнув правой кнопкой мыши на имени нужной переменной в панели **Переменные** и выбрав в появившемся на экране контекстном меню пункт **Добавить контрольное значение**. Также можно щелкнуть на значке **Добавить выражение**, расположенном в заголовке раздела **Контрольное значение**, ввести имя нужной переменной в появившееся поле ввода и нажать клавишу `<Enter>`. Занесем в этот раздел переменные `i` и `j`.

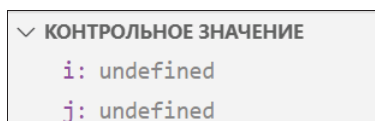


Рис. 13.5. Раздел **Контрольное значение** панели **Запуск и отладка**

Для пошагового выполнения программы предназначены следующие пункты в меню **Выполнить** или соответствующие кнопки на небольшой панели инструментов, которая появится сверху после запуска отладки:

- **Шаг с заходом** (клавиша `<F11>`) — выполняет одну инструкцию. Сделаем два шага и посмотрим на значения переменных `i` и `j` в разделе **Контрольное значение** — обе переменные получили значение 0. Делаем еще четыре шага и останавливаемся на строке 12, которая у нас помечена точкой останова. Если мы сделаем еще один шаг, то попадем внутрь функции `print()`. Обратите внимание на значения переменных `i` и `j` в окне **Контрольное значение** — они стали не определены, т. к. мы находимся в области видимости функции `print()`;
- **Шаг с выходом** (комбинация клавиш `<Shift>+<F11>`) — позволяет сразу выйти из функции. Выйдем из функции `print()` — в результате текущей станет строка 11;
- **Шаг с обходом** (клавиша `<F10>`) — выполняет одну инструкцию. Если в инструкции осуществляется вызов функции, то функция выполняется и отладчик переходит в режим ожидания после выхода из функции. Сделаем три шага и опять окажемся на строке 11, не входя в функцию `print()`;
- **Продолжить** (клавиша `<F5>`) — выполняет инструкции до следующей точки останова или до конца программы при отсутствии точек. Нажимая клавишу `<F5>`, мы каждый раз будем останавливаться на строке 12 и сможем наблюдать за текущими значениями переменных `i` и `j` в разделе **Контрольное значение**, а также за выводом результатов работы программы на панели **Консоль отладки**. На одном из шагов можно обнаружить, что значение переменной `i` стало равно 2, и сразу после следующего шага возникает исключение `TypeError`. Если вы не

успели этого заметить, запустите отладку еще раз и наблюдайте за значениями переменных. Так как значение переменной `i` стало равно 2, а в следующей строке идет обращение к массиву по этому индексу и возникает исключение, следовательно, что-то не так со значением переменной `i`. Смотрим внимательно на предыдущую инструкцию и обнаруживаем неправильное условие окончания цикла в строке 10 (`i <= arr.length`).

ОБРАТИТЕ ВНИМАНИЕ

В описании исключения стоит строка с номером 11, а реальная ошибка в строке 10.

Исправляем ошибку, и выражение примет вид `i < arr.length`. Опять запускаем отладку, чтобы убедиться в правильности внесенных изменений.

Если мы попробуем перейти сразу к строке 14, то это не получится, т. к. существует точка останова на строке 12. А нам бы хотелось увидеть вывод одной строки массива сразу. Для этого можно временно отключить точку останова, щелкнув на ней правой кнопкой мыши и выбрав в контекстном меню пункт **Отключить**. Можно временно отключить сразу все точки останова, щелкнув на значке **Включить или отключить точки останова** в заголовке раздела **Точки останова**.

Отключим точку останова на строке 12 и добавим точку на строке 14. Теперь, нажав клавишу `<F5>`, можно сразу перейти к строке 14. После этого действия мы заметим, что вместо четырех значений выводятся только три. Чтобы понять причину, опять выполняем программу по шагам и наблюдаем за значениями переменных в разделе **Контрольное значение**. Наконец, вспоминаем, что индексы нумеруются с 0, а не с 1, и исправляем ошибку в строке 11 (`j = 0` вместо `j = 1`);

- ❑ **Перезапустить** (комбинация клавиш `<Ctrl>+<Shift>+<F5>`) — перезапускает отладчик, что может понадобиться после правки программного кода;
- ❑ **Остановить отладку** (комбинация клавиш `<Shift>+<F5>`) — останавливает отладку.

Применение отладки — это самый эффективный способ нахождения ошибок, не требующий вставки в код никаких инструкций для вывода промежуточных значений. На каждом шаге мы можем наблюдать за значениями всех или только избранных переменных. После отладки не нужно удалять или временно отключать какие-либо инструкции. Даже точки останова убирать не нужно — при обычном выполнении они ни на что не влияют. Пользуйтесь отладкой при возникновении любой ошибки — и вы очень быстро ее найдете и исправите.

13.8. Строгий режим

Чтобы уменьшить вероятность возникновения ошибок, следует перевести программу в *строгий режим*. Для этого в начало программы или в начало модуля достаточно добавить инструкцию:

```
'use strict';
```


или:

```
"use strict";
```

При этом все подключаемые модули также переводятся в строгий режим, что может привести к ошибкам, если код какого-либо модуля не соответствует правилам строгого режима. Это не относится к модулям ECMAScript, код которых всегда выполняется в строгом режиме. Чтобы избежать проблем со сторонним кодом, строгий режим лучше включать в начале функций, например:

```
function func() {  
    'use strict';  
    // Код, выполняемый в строгом режиме  
}
```

Если код не соответствует правилам строгого режима, то при запуске программы генерируется исключение и выполнение программы прерывается. Благодаря этому вы узнаете о проблеме с кодом еще при первом запуске программы и сможете сразу исправить ошибки. Рассмотрим основные отличия между режимами.

В обычном режиме можно присвоить значение переменной, которая не была объявлена явным образом:

```
x = 10;  
console.log('x =', x); // x = 10
```

При использовании строгого режима будет сгенерировано исключение:

```
'use strict';  
x = 10; // ReferenceError: x is not defined
```

Исключение также возникнет при попытке создать глобальную переменную внутри функции:

```
function func() {  
    'use strict';  
    x = 88; // OK. Существует глобальная переменная  
    y = 34; // ReferenceError: y is not defined  
}  
let x = 3; // Глобальная переменная  
func();
```

В обычном режиме можно присвоить значения ключевым словам языка, что не приведет ни к какому результату:

```
undefined = 'und';  
console.log(undefined); // undefined  
NaN = 0;  
console.log(NaN); // NaN  
Infinity = -1;  
console.log(Infinity); // Infinity
```


При использовании строгого режима будет сгенерировано исключение:

```
undefined = 'und';
// TypeError: Cannot assign to read only property
// 'undefined' of object '#<Object>'
NaN = 0;
// TypeError: Cannot assign to read only property
// 'NaN' of object '#<Object>'
Infinity = -1;
// TypeError: Cannot assign to read only property
// 'Infinity' of object '#<Object>'
```

В строгом режиме попытка объявления переменной, имя которой совпадает с зарезервированным ключевым словом, приведет к исключению:

```
'use strict';
let implements = 0;
// SyntaxError: Unexpected strict mode reserved word
let interface = 0;
// SyntaxError: Unexpected strict mode reserved word
let package = 0;
// SyntaxError: Unexpected strict mode reserved word
let private = 0;
// SyntaxError: Unexpected strict mode reserved word
let protected = 0;
// SyntaxError: Unexpected strict mode reserved word
let public = 0;
// SyntaxError: Unexpected strict mode reserved word
let static = 0;
// SyntaxError: Unexpected strict mode reserved word
let yield = 0;
// SyntaxError: Unexpected strict mode reserved word
(function() {
  'use strict';
  var let = 5;
  // SyntaxError: Unexpected strict mode reserved word
})();
```

В обычном режиме восьмеричное значение может начинаться не только с комбинации 0o, но и просто с нуля. В строгом режиме второй вариант приведет к исключению. Пример:

```
'use strict';
console.log(0o167); // OK. 119
console.log(0167);
// SyntaxError: Octal literals are not allowed in strict mode.
```

Существуют особенности при использовании функции `eval()`. Переменные, объявленные внутри выражения, в обычном режиме импортируются в область видимости указателя `this` внутри функции `eval()`. Если функция `eval()` вызывается непосред-

ственно внутри пользовательской функции, то будут перезаписаны значения локальных переменных, а если через ссылку (указатель `this` при этом ссылается на глобальную область видимости) — могут быть затронуты значения глобальных переменных. Пример:

```
var x = 10;
let str = 'var x = 3;';
eval(str);
console.log(x);           // 3
```

В строгом режиме переменные не импортируются вообще:

```
'use strict';
var x = 10;
let str = 'var x = 3; x';
let y = eval(str);
console.log(x);           // 10
console.log(y);           // 3
```

Если первый параметр метода `call()` имеет значение `undefined` или `null`, то в обычном режиме функция вызывается в глобальном контексте:

```
function sum(x, y) {
  console.log(this); // Object [global] {...}
  return x + y;
}
console.log( sum.call(null, 5, 7) ); // 12
```

В строгом режиме в этом случае указатель `this` будет иметь значение `undefined` или `null`:

```
'use strict';
function sum(x, y) {
  console.log(this); // undefined или null
  return x + y;
}
console.log( sum.call(undefined, 5, 7) ); // 12 this === undefined
console.log( sum.call(null, 5, 5) );      // 10 this === null
```

Аналогичная ситуация с методом `apply()` и одноименным статическим методом из класса `Reflect`:

```
console.log( sum.apply(undefined, [5, 7]) ); // 12 this === undefined
console.log( sum.apply(null, [5, 5]) );      // 10 this === null
console.log( Reflect.apply(sum, undefined, [5, 7]) );
// 12 this === undefined
console.log( Reflect.apply(sum, null, [5, 5]) );
// 10 this === null
```

При простом вызове функции в строгом режиме указатель `this` будет иметь значение `undefined`:

```
console.log( sum(5, 7) ); // 12 this === undefined
```

Если не указан контекст в методе `bind()` или мы сохраняем ссылку на метод объекта в переменной, то `this` будет иметь значение `undefined`:

```
'use strict';
let obj = {
  func: function() {
    console.log(this);
  }
};
let fn = obj.func;
fn();      // this === undefined в строгом режиме
let fnBind = fn.bind(obj);
fnBind(); // this === obj
fnBind = fn.bind();
fnBind(); // this === undefined в строгом режиме
fnBind = fn.bind(undefined);
fnBind(); // this === undefined в строгом режиме
fnBind = fn.bind(null);
fnBind(); // this === null в строгом режиме
```

В строгом режиме внутри пользовательской функции нельзя использовать устаревшее свойство `callee` объекта `arguments`. Это свойство следует заменять на имя функции. Пример:

```
'use strict';
let fn = function test(x, y) {
  console.log( test );           // OK. [Function: test]
  console.log( arguments.callee ); // Error
  // TypeError: 'caller', 'callee', and 'arguments' properties
  // may not be accessed on strict mode functions or the
  // arguments objects for calls to them
}
fn(5, 6);
```

Если в объявлении функции присутствуют одноименные параметры, то в обычном режиме при вызове функции параметр получит последнее переданное значение. В строгом режиме наличие одноименных параметров приведет к исключению. Пример:

```
'use strict';
function func(a, a) {
  console.log(a);      // В обычном режиме: a = 20
}
func(10, 20);
// SyntaxError: Duplicate parameter name not allowed in this context
```

В строгом режиме попытка изменения значения свойства, доступного только для чтения, приведет к исключению, тогда как в обычном режиме присваивание нового значения не приводит ни к какому результату:

```
'use strict';
let obj = Object.create(Object.prototype, {
  'a': { value: 10, writable: false }
});
obj.a = 17;
// TypeError: Cannot assign to read only property 'a'
// of object '#<Object>'
```

В строгом режиме попытка удаления свойства, атрибут `configurable` которого имеет значение `false`, приведет к исключению, тогда как в обычном режиме операция не приводит ни к какому результату:

```
'use strict';
let obj = Object.create(Object.prototype, {
  'a': { value: 10, configurable: false }
});
delete obj.a;
// TypeError: Cannot delete property 'a' of #<Object>
```

Аналогичная ситуация возникнет при попытке удаления переменной:

```
'use strict';
let a = 10;
delete a;
// SyntaxError: Delete of an unqualified identifier in strict mode.
```

Использование строгого режима позволит избежать различных ошибок. Старайтесь всегда работать в строгом режиме, хотя бы на этапе написания кода. При этом учитывайте, что код модулей ECMAScript всегда выполняется в строгом режиме, поэтому добавлять инструкцию `"use strict"`; явным образом не нужно.

13.9. Установка и настройка *Prettier*

Как уже говорилось, форматирование кода позволит найти ошибку гораздо быстрее. Чтобы в редакторе VS Code отформатировать все содержимое вкладки, в контекстном меню выбираем пункт **Форматировать документ** или нажимаем комбинацию клавиш `<Shift>+<Alt>+<F>`. Можно также выделить фрагмент и в контекстном меню выбрать пункт **Форматировать выделенный фрагмент** или нажать комбинацию клавиш `<Ctrl>+<K>`, а затем `<Ctrl>+<F>`.

В настройках редактора можно найти много пунктов, задающих отдельные настройки форматирования. Если установлен флажок **Editor: Format On Paste**, будет выполняться автоматическое форматирование кода при вставке из буфера обмена. Если установлен флажок **Editor: Format On Save**, код будет форматироваться при сохранении.

Также можно установить расширение Prettier – Code formatter. Не исключено, что результат его работы понравится вам больше, чем выдаваемый VS Code. Открываем панель **Расширения**, нажав комбинацию клавиш `<Shift>+<Ctrl>+<X>`, в строке

поиска вводим "Prettier", находим расширение в списке и устанавливаем его. После установки в строке состояния появится значок с надписью **Prettier**. Щелчок на этом значке откроет нижнюю панель на вкладке **Выходные данные**, куда выводится подробная информация о процессе форматирования и используемых настройках.

Дополнительно следует установить пакет Prettier для Node.js. Для этого в командной строке выполняем следующую команду:

```
npm install prettier --save-dev --save-exact
```

После установки в раздел `devDependencies` внутри файла `package.json` будет добавлена строка:

```
"prettier": "2.2.1"
```

После установки щелкаем правой кнопкой мыши в области вкладки и из контекстного меню выбираем пункт **Форматировать документ с помощью**. Откроется панель **Палитра команд** со списком доступных средств для форматирования кода. Из списка выбираем пункт **Prettier – Code formatter** — и документ будет отформатирован.

В настройках редактора можно указать способ форматирования по умолчанию. Для этого в файл `settings.json` (расположен в каталоге `C:\Users\<Имя пользователя>\AppData\Roaming\Code\User` или в каталоге `.vscode` текущего пакета) добавляем следующие строки:

```
"[javascript]": {
  "editor.defaultFormatter": "esbenp.prettier-vscode",
  "editor.formatOnSave": true,
  "editor.formatOnPaste": true
},
"files.autoSave": "off"
```

Дополнительно отключаем автоматическое сохранение файла, изменяя существующую настройку параметра `files.autoSave`, иначе вы не сможете вставить что-либо из буфера обмена с помощью контекстного меню. Представьте, вставляете пустую строку, отображаете контекстное меню, вкладка теряет фокус, срабатывает автоматическое сохранение, форматирование, и пустая строка удалится еще до выбора пункта меню. Так что придется выбирать что-то одно: автоматическое сохранение или автоматическое форматирование при сохранении файла. Напомним, что сохранить файл позволяет комбинация клавиш `<Ctrl>+<S>`.

Параметр `editor.defaultFormatter` задает способ форматирования по умолчанию при выборе из контекстного меню пунктов **Форматировать документ** или **Форматировать выделенный фрагмент**. Значение `"esbenp.prettier-vscode"` означает, что форматирование по умолчанию будет реализовано расширением Prettier. Дополнительно мы указали параметр `editor.formatOnSave` со значением `true`, который задает автоматическое форматирование кода при сохранении файла, и параметр `editor.formatOnPaste` со значением `true`, который включает форматирование при вставке фрагмента из буфера обмена.

Все параметры расположены внутри раздела `[javascript]`, что означает применение этих параметров только к JavaScript-коду. Если нужно применить эти параметры к другим языкам, следует создать новый раздел с названием языка. Если настройки указаны вне разделов, они применяются ко всем языкам. Следует учесть, что Prettier поддерживает не все языки. Вот примерный список языков в базовом пакете:

```
Enabling prettier for languages: ansible, css, graphql, home-assistant,  
html, javascript, javascriptreact, json, json5, jsonc, less, markdown,  
mdx, mongo, postcss, scss, typescript, typescriptreact, vue, yaml
```

Некоторые параметры форматирования можно настроить с помощью конфигурационного файла `.prettierrc`, расположенного в корне пакета. Создадим в каталоге `C:\book\p1` файл `.prettierrc` с содержимым из листинга 13.4.

Листинг 13.4. Файл `.prettierrc`

```
{  
  "tabWidth": 3,  
  "endOfLine": "lf",  
  "printWidth": 80,  
  "semi": true,  
  "singleQuote": true,  
  "useTabs": false,  
  "quoteProps": "as-needed",  
  "bracketSpacing": true,  
  "arrowParens": "avoid",  
  "requirePragma": false,  
  "insertPragma": false,  
  "trailingComma": "none",  
  "htmlWhitespaceSensitivity": "css",  
  "proseWrap": "preserve",  
  "jsxBracketSameLine": false,  
  "jsxSingleQuote": false,  
  "vueIndentScriptAndStyle": false,  
  "overrides": [  
    {  
      "files": ["*.html", "*.css", "*.scss"],  
      "options": {  
        "tabWidth": 2  
      }  
    }  
  ]  
}
```

Рассмотрим основные параметры внутри файла `.prettierrc` (полный список параметров и имен конфигурационного файла можно найти в документации):

- ❑ `parser` — задает анализатор файла. Обычно значение определяется автоматически по расширению файла. Полный список допустимых значений можно найти в документации. Для JavaScript-файлов по умолчанию используется значение `"babel"`;
- ❑ `filepath` — задает путь к файлу, например: `"c:\\book\\p1\\index.js"`;
- ❑ `tabWidth` — число пробелов для создания отступа. Значение по умолчанию: 2. Советуем установить значение 3 — для JavaScript-файлов и 2 — для HTML-, CSS- и SCSS-файлов с помощью параметра `overrides`;
- ❑ `useTabs` — если указано значение `true`, то отступы выполняются с помощью табуляции, а если `false` — то с помощью пробелов. Советуем установить значение `false`;
- ❑ `endOfLine` — символы перевода строк. Допустимые значения: `"auto"` (автоматическое определение по первой строке), `"lf"` (`\n` — значение по умолчанию и рекомендуемое значение), `"crlf"` (`\r\n`) и `"cr"` (`\r`);
- ❑ `printWidth` — предпочтительная длина строки в символах. Значение по умолчанию и рекомендуемое значение — 80;
- ❑ `semi` — если указано значение `true`, то точка с запятой должна быть после каждой инструкции, а если `false` — только по мере необходимости (если точка с запятой не нужна, она будет удалена). Значение по умолчанию и рекомендуемое значение — `true`;
- ❑ `singleQuote` — если указано значение `true`, будут использоваться одинарные кавычки, а если `false` — только двойные. Значение по умолчанию: `false`. Рекомендуемое значение — `true`.

Если указано значение `true`, то следующие инструкции:

```
let str = "строка";  
let str2 = "Д'Арк";  
let str3 = "Группа \"Кино\"";
```

будут преобразованы так:

```
let str = 'строка';  
let str2 = "Д'Арк";  
let str3 = 'Группа "Кино"';
```

Как видно из результата, двойные кавычки заменяются одинарными только в том случае, если внутри строки нет одинарных кавычек;

- ❑ `quoteProps` — правила использования кавычек в именах свойств объектов. Допустимые значения: `"as-needed"` — кавычки добавляются только там, где это необходимо (значение по умолчанию и рекомендуемое значение), `"consistent"` — если имя хотя бы одного свойства указано в кавычках, то в кавычки будут заключаться имена всех свойств, и `"preserve"` — кавычки не будут ни добавляться, ни убираться автоматически.

Если указано значение "as-needed", то следующая инструкция:

```
let obj = {  
  "a": 5,  
  "b-2": 7  
};
```

будет преобразована так:

```
let obj = {  
  a: 5,  
  'b-2': 7  
};
```

Если указано значение "consistent", то следующая инструкция:

```
let obj = {  
  a: 5,  
  "b-2": 7  
};
```

будет преобразована так:

```
let obj = {  
  'a': 5,  
  'b-2': 7  
};
```

Если указано значение "preserve", то следующая инструкция:

```
let obj = {  
  a: 5,  
  "b-2": 7,  
  'c': 2  
};
```

будет преобразована так:

```
let obj = {  
  a: 5,  
  'b-2': 7,  
  'c': 2  
};
```

- `bracketSpacing` — если указано значение `true`, то после открывающей и перед закрывающей фигурными скобками литерала объекта будут добавлены пробелы, а если `false` — то пробелы не добавляются. Значение по умолчанию и рекомендуемое значение — `true`.

Если указано значение `true`, то следующая инструкция:

```
let obj = {a:5};
```

будет преобразована так:

```
let obj = { a: 5 };
```


- ❑ `arrowParens` — управляет расстановкой круглых скобок в определении стрелочной функции с одним параметром. Допустимые значения: `"always"` — круглые скобки ставятся (значение по умолчанию) и `"avoid"` — круглые скобки не ставятся.

Если указано значение `"always"`, то следующая инструкция:

```
let print = msg => console.log(msg);
```

будет преобразована так:

```
let print = (msg) => console.log(msg);
```

- ❑ `trailingComma` — управляет расстановкой завершающих запятых у свойств в литералах объектов. Допустимые значения: `"none"` — завершающая запятая никогда не ставится (значение по умолчанию), `"es5"` — завершающие запятые ставятся, если они допустимы в ES5, и `"all"` — завершающие запятые везде, где они допустимы.

Если указано значение `"es5"` или `"all"`, то следующие инструкции:

```
let obj = {  
  a: 5,  
  b: 2  
};  
let arr = [  
  [1, 2, 3, 4],  
  [5, 6, 7, 8]  
];
```

будут преобразованы так:

```
let obj = {  
  a: 5,  
  b: 2,  
};  
let arr = [  
  [1, 2, 3, 4],  
  [5, 6, 7, 8],  
];
```

Зачем добавлять завершающую запятую? Во-первых, при добавлении новой строки не будет ошибки, т. к. вы не забудете поставить запятую — ведь она уже стоит. Во-вторых, это очень полезно при использовании Git: если поставите запятую и добавите новую строку, то в изменения попадет лишняя строка, помимо добавленной (та, в которой вы поставили запятую), а если запятая будет стоять заранее, то в изменения попадет только добавленная строка;

- ❑ `overrides` — массив с объектами, содержащими настройки для указанных расширений файлов. Свойство `files` внутри объекта в качестве значения принимает массив строк с шаблонами сопоставления, а свойство `options` — объект с параметрами. В нашем примере мы переопределяем значение параметра `tabWidth` для HTML-, CSS- и SCSS-файлов:

```
"overrides": [  
  {  
    "files": ["*.html", "*.css", "*.scss"],  
    "options": {  
      "tabWidth": 2  
    }  
  }  
]
```

Prettier не всегда справляется так, как нам хочется. Если нужно отключить автоматическое форматирование для одной инструкции, то перед инструкцией нужно добавить комментарий с текстом `prettier-ignore`:

```
// prettier-ignore  
let str = "Группа \"Кино\"";
```

Иногда таких комментариев становится слишком много, и некоторые программисты отказываются от использования Prettier. Ведь настроек у него слишком мало. Причем сделано это специально, поэтому ждать увеличения количества настроек не приходится. Тем не менее для начинающих программистов Prettier очень полезен. Кроме того, Prettier незаменим для работы в команде, в которой участвуют программисты с разным уровнем подготовки, — он отформатирует код в едином стандарте.

Помимо работы внутри редактора VS Code, Prettier позволяет запускать форматирование файлов из командной строки. Для начала отключим автоматическое форматирование при сохранении файла, внося следующую корректировку в файл `settings.json`:

```
"[javascript]": {  
  "editor.defaultFormatter": "esbenp.prettier-vscode",  
  "editor.formatOnSave": false  
}
```

Дополнительно в корне пакета нужно создать файл `.prettiignore`, внутри которого записать файлы и каталоги, содержимое которых преобразовывать не нужно. Содержимое каталога `node_modules` всегда игнорируется, за исключением явного указания флага `--with-node-modules`.

Команда запуска Prettier из командной строки имеет такой формат:

```
prettier <Опции> <Файл, каталог или шаблон>
```

Чтобы просто проверить правильность форматирования без изменения содержимого файлов, следует указать флаг `--check`. Для форматирования файлов предназначен флаг `--write`. Содержимое файлов при этом будет перезаписано. Полный список опций можно найти в документации.

Давайте внутри секции `scripts` в файле `package.json` добавим два скрипта:

```
"format": "prettier --write \"**/*.js\"",  
"format:check": "prettier --check \"**/*.js\""
```

Здесь мы указали шаблон, который соответствует всем файлам с расширением `js`, расположенным в текущем каталоге и во вложенных каталогах. Откроем командную строку или терминал в редакторе и запустим скрипт проверки:

```
C:\book\pl>npm run format:check
```

Примерный результат при наличии ошибок будет выглядеть так:

```
Checking formatting...
[warn] index.js
[warn] module1.js
[warn] modules\module1.js
[warn] test.js
[warn] Code style issues found in the above file(s). ㄹ
Forgot to run Prettier?
```

В случае наличия ошибок в файлах их имена выводятся в консоли. Кроме того, в каталоге `C:\Users\<Имя пользователя>\AppData\Roaming\npm-cache\_logs` создается файл с полным описанием ошибки, но создает его не Prettier (он просто выбрасывает код ошибки 1), а NPM, который на этот код так реагирует. Не забывайте периодически чистить этот каталог.

Теперь запустим скрипт `format`:

```
C:\book\pl>npm run format
```

```
> pl@1.0.0 format C:\book\pl
> prettier --write "**/*.js"
```

```
index.js 87ms
module1.js 12ms
modules\module1.js 6ms
test.js 10ms
```

Все файлы были отформатированы. Проверим:

```
C:\book\pl>npm run format:check
```

```
> pl@1.0.0 format:check C:\book\pl
> prettier --check "**/*.js"
```

```
Checking formatting...
All matched files use Prettier code style!
```

Проверка прошла успешно.

Теперь предположим, что нам нужно дополнительно проверить и отформатировать какие-либо другие файлы, например файл `myJSON.json`. Для этого после команды добавляем три дефиса, а затем передаем имя файла:

```
C:\book\pl>npm run format:check --- myJSON.json
```

Теперь отформатируем все файлы с расширением js и файл myJSON.json:

```
C:\book\pl>npm run format --- myJSON.json
```

```
> pl@1.0.0 format C:\book\pl
> prettier --write "**/*.js" "myJSON.json"
index.js 81ms
module1.js 10ms
modules\module1.js 5ms
myJSON.json 9ms
```

13.10. Установка и настройка *ESLint*

Существует еще одно средство, выявляющее ошибки в коде, не дожидаясь сохранения файла, — ESLint, интегрирующееся с Prettier. Давайте установим и настроим его.

Сначала установим расширение ESLint для редактора VS Code. Как обычно, переходим в панель **Расширения**, нажав комбинацию клавиш <Shift>+<Ctrl>+<X>, в строке поиска вводим "ESLint", находим расширение в списке и устанавливаем его.

Дополнительно следует установить пакет ESLint для Node.js. Для этого в командной строке выполняем следующую команду:

```
npm install eslint --save-dev
```

Кроме того, понадобится набор готовых правил, входящий в состав пакета eslint-config-airbnb-base, и пакет eslint-plugin-import. Устанавливаем пакеты с помощью команды:

```
npm install eslint-config-airbnb-base eslint-plugin-import --save-dev
```

Для связки ESLint с Prettier последний должен быть установлен. Если вы не выполняли установку по инструкциям из предыдущего раздела, то самое время подать команду установки Prettier:

```
npm install prettier --save-dev --save-exact
```

Дополнительно нужно установить пакеты eslint-config-prettier и eslint-plugin-prettier:

```
npm install eslint-config-prettier eslint-plugin-prettier --save-dev
```

После установки в раздел devDependencies внутри файла package.json будут добавлены следующие строки:

```
"eslint": "^7.25.0",
"eslint-config-airbnb-base": "^14.2.1",
"eslint-config-prettier": "^8.3.0",
"eslint-plugin-import": "^2.22.1",
"eslint-plugin-prettier": "^3.4.0",
"prettier": "2.2.1"
```

Далее в каталоге `C:\book\p1` создаем конфигурационный файл `.eslintrc.json` с содержанием, приведенным в листинге 13.5.

Листинг 13.5. Файл `.eslintrc.json`

```
{
  "extends": ["airbnb-base", "prettier"],
  "plugins": ["prettier"],
  "env": {
    "es6": true,
    "commonjs": true,
    "node": true
  },
  "rules": {
    "prettier/prettier": "error",
    "no-plusplus": "off",
    "prefer-const": "off",
    "no-unused-vars": ["warn", { "argsIgnorePattern": "val|req|res" }],
    "strict": "off",
    "semi": "error",
    "no-console": "off",
    "max-len": ["error", { "code": 80, "ignoreUrls": true }],
    "quotes": [
      "error",
      "single",
      { "avoidEscape": true, "allowTemplateLiterals": false }
    ]
  }
}
```

В качестве значения свойства `extends` указываем пакеты с конфигурацией без префикса `eslint-config-`:

```
"extends": ["airbnb-base", "prettier"]
```

В следующей инструкции настраиваем связку с Prettier, подключая пакет `eslint-plugin-prettier`:

```
"plugins": ["prettier"]
```

Все ошибки форматирования, сгенерированные этим пакетом, делаем ошибками ESLint, добавляя в раздел `rules` следующую инструкцию:

```
"prettier/prettier": "error"
```

Раздел `env` задает настройки конфигурации. Свойство `es6` устанавливает правила для стандарта ES6. По умолчанию используются правила ES5, поэтому ключевые слова `let` и `const` без явного указания свойства `es6` проверку не проходят. Свойство `commonjs` включает поддержку модулей CommonJS, а свойство `node` — поддержку Node.js.

Раздел `rules` описывает правила проверки кода. Этих правил очень много, и описать их в настоящей книге не представляется возможным. Полное описание правил приведено в документации. Чаще всего свойства с названиями правил содержат одно из трех значений:

- ❑ `"off"` (или `0`) — отключает проверку правила. Например, так можно отключить проверку наличия вывода в консоль:

```
"no-console": "off"
```

- ❑ `"warn"` (или `1`) — при нарушении правила выводится предупреждающее сообщение, а фрагмент кода подчеркивается желтой волнистой линией. Например, выведем предупреждающее сообщение, если переменная объявлена, но не используется в программе:

```
"no-unused-vars": "warn"
```

- ❑ `"error"` (или `2`) — при нарушении правила выводится сообщение об ошибке, а фрагмент кода подчеркивается красной волнистой линией. Например, выведем сообщение об ошибке, если после инструкции не указана точка с запятой:

```
"semi": "error"
```

В некоторых случаях можно перечислить дополнительные условия, указав в качестве значения массив с настройками. Например, установим максимальную длину строки в 80 символов, но проигнорируем длинные URL:

```
"max-len": ["error", { "code": 80, "ignoreUrls": true }]
```

Выведем предупреждение, если переменная объявлена и не используется, но при этом проигнорируем параметры функций с именами `val`, `req` и `res`:

```
"no-unused-vars": ["warn", { "argsIgnorePattern": "val|req|res" }]
```

Откроем любой существующий JavaScript-файл (например, `C:\book\p1\module1.js`) и попробуем записать в него такую инструкцию:

```
let str = 'строка';
```

В результате имя переменной `str` будет подчеркнуто желтой волнистой линией, что соответствует предупреждающему сообщению. Если навести указатель мыши на имя переменной, отобразится всплывающее окно с сообщением об ошибке:

```
'str' is assigned a value but never used. eslint (no-unused-vars)
```

Аналогичное сообщение можно увидеть на вкладке **Проблемы**, расположенной на нижней панели редактора VS Code. В левой части статусной строки отображается количество ошибок и количество предупреждений. Достаточно щелкнуть левой кнопкой мыши на этих значках — и отобразится содержимое вкладки **Проблемы**. Держите эту вкладку открытой — и сразу увидите сообщения обо всех найденных ESLint проблемах.

Что делать с этим предупреждающим сообщением? Во-первых, можно полностью отключить проверку, дав значение `"off"` правилу `no-unused-vars`. Однако такое решение неправильное.

ОБРАТИТЕ ВНИМАНИЕ

Название правила указано внутри круглых скобок в составе текста сообщения.

Во-вторых, можно просто удалить эту инструкцию, если она лишняя. В-третьих, можно указать переменную `str` в других инструкциях:

```
let str = 'строка';  
console.log(str);
```

Предупреждающее сообщение исчезнет. В-четвертых, можно закомментировать инструкцию, если планируется использовать ее позже. В-пятых, можно написать комментарий с текстом `eslint-disable-next-line` перед инструкцией для временно-го отключения проверки:

```
// eslint-disable-next-line  
let str = 'строка';
```

Можно перечислить через запятую несколько отключаемых правил:

```
// eslint-disable-next-line no-unused-vars, semi, prettier/prettier  
let str = 'строка'
```

Комментарий может следовать после инструкции. В этом случае текст должен быть `eslint-disable-line`:

```
let str = 'строка'; // eslint-disable-line no-unused-vars
```

Если инструкций несколько, то начало фрагмента помечается примечанием с текстом `eslint-disable`, а конец — примечанием с текстом `eslint-enable`:

```
/* eslint-disable no-unused-vars */  
let str = 'строка';  
let x = 5;  
/* eslint-enable */
```

Во всплывающем окне помимо сообщения об ошибке расположена ссылка **Быстрое исправление**. Если щелкнуть на ней, отобразится список с возможными действиями. Например, не поставим точку с запятой в конце инструкции:

```
console.log('Текст сообщения')
```

В результате закрывающая круглая скобка будет подчеркнута красной волнистой линией. Наводим указатель мыши на подчеркивание и во всплывающем окне щелкаем на ссылке **Быстрое исправление**. Чтобы автоматически исправить проблему, из меню выбираем пункт **Fix this <Правило> problem** — точка с запятой будет добавлена автоматически. Если нужно исправить сразу несколько проблем, выбираем пункт **Fix all <Правило> problems**. Из этого же списка можно выбрать пункты, вставляющие комментарии, которые отключают проверку, а также пункты со ссылками на документацию.

Чтобы проблемы решались автоматически при сохранении файла, вносим следующую корректировку в файл конфигурации `settings.json`:

```
"[javascript]": {  
  "editor.defaultFormatter": "esbenp.prettier-vscode",
```

```
"editor.formatOnSave": false,  
"editor.codeActionsOnSave": {  
  "source.fixAll.eslint": true  
},  
"files.autoSave": "off"
```

Теперь при сохранении файла будет предпринята попытка исправления ошибок, которые можно автоматически исправить, а также выполнено форматирование кода с помощью Prettier. Одним выстрелом мы убили сразу двух зайцев.

Помимо работы внутри редактора VS Code, ESLint позволяет запускать проверку файлов из командной строки. Для начала отключим автоматическую проверку при сохранении файла, внося корректировку в файл `settings.json`:

```
"source.fixAll.eslint": false
```

Дополнительно в корне пакета нужно создать файл `.eslintignore`, внутри которого прописать файлы и каталоги, содержимое которых преобразовывать не нужно. Содержимое каталога `node_modules`, а также файлов и каталогов, имена которых начинаются с точки, всегда игнорируется.

Команда запуска ESLint из командной строки имеет такой формат:

```
eslint <Опции> <Файл, каталог или шаблон>
```

Чтобы просто проверить правильность форматирования без изменения содержимого файлов, следует запустить команду без флагов. Для исправления ошибок внутри файлов используется флаг `--fix`. Содержимое файлов при этом будет перезаписано. Полный список опций можно найти в документации.

Давайте внутри секции `scripts` в файле `package.json` добавим два скрипта:

```
"lint": "eslint \"**/*.js\"",  
"lint:fix": "eslint --fix --color \"**/*.js\""
```

Здесь мы указали шаблон, который соответствует всем файлам с расширением `js`, расположенным в текущем каталоге и во вложенных каталогах. Откроем командную строку или терминал в редакторе и запустим скрипт проверки:

```
C:\book\pl>npm run lint
```

Примерный результат при наличии ошибок будет выглядеть так:

```
C:\book\pl\index.js  
1:7   warning  'process' is assigned a value but never used  
no-unused-vars  
1:25   error     Unable to resolve path to module 'process2'  
import/no-unresolved  
3:5   warning  'str' is assigned a value but never used  
no-unused-vars  
4:13   error     Replace `\"Текст сообщения\"` with `\"Текст сообщения\"`;  
prettier/prettier  
4:13   error     Strings must use singlequote
```



```
quotes
  4:31  error    Missing semicolon
semi
  6:5   warning  'y' is assigned a value but never used
no-unused-vars
```

При наличии ошибок в файлах их имена выводятся в консоли. Кроме того, в каталоге `C:\Users\<Имя пользователя>\AppData\Roaming\npm-cache\_logs` появится файл с полным описанием ошибки, но создает его не ESLint (он просто выбрасывает код ошибки 1), а NPM, который на этот код так реагирует. Не забывайте периодически чистить этот каталог.

Теперь запустим скрипт `lint:fix`:

```
C:\book\p1>npm run lint:fix
```

```
> p1@1.0.0 lint:fix C:\book\p1
> eslint --fix --color "**/*.js"
```

```
C:\book\p1\index.js
  1:7   warning  'process' is assigned a value but never used
no-unused-vars
  1:25  error      Unable to resolve path to module 'process2'
import/no-unresolved
  3:5   warning  'str' is assigned a value but never used
no-unused-vars
  6:5   warning  'y' is assigned a value but never used
no-unused-vars

  4 problems (1 error, 3 warnings)
```

Все ошибки, которые можно было исправить, успешно исправлены во всех файлах. Кроме того, файлы были отформатированы с помощью Prettier. Согласитесь, что это очень удобно.



ГЛАВА 14

Асинхронность

До этой главы в большинстве случаев инструкции в программе выполнялись последовательно, строго одна за другой, т. е. синхронно. Выполнение следующей инструкции начиналось только после завершения выполнения предыдущей. Однако в ряде случаев (например, работа с файлами, загрузка данных из сети и др.) инструкции в Node.js могут выполняться асинхронно, когда следующая инструкция начинает выполняться еще до того, как закончится выполнение предыдущей. Мы уже встречались с асинхронным выполнением кода при изучении таймеров в *разд. 9.7*:

```
console.log('До таймера');
setTimeout( function(x) {
    console.log('Интервал истек. x =', x);
}, 2000, 15);
console.log('Конец программы');
```

Результат в консоли (последняя строка отобразится через две секунды, а не сразу):

```
До таймера
Конец программы
Интервал истек. x = 15
```

Во многих языках программирования при достижении конца программы все потоки и сама программа завершаются. В Node.js это не так. После выполнения последней инструкции интерпретатор ожидает завершения всех работающих асинхронных задач и только после этого останавливает работу программы.

14.1. Объект класса *Promise*

Для получения результата выполнения асинхронных операций предназначен объект класса `Promise`.

14.1.1. Создание объекта класса *Promise*

Создать объект `Promise` позволяет следующий конструктор:

```
const myPromise = new Promise( function(resolve, reject) {
    // ...
});
```

Конструктор принимает ссылку на функцию с двумя параметрами, также являющимися ссылками на функции:

- `resolve(<Данные>)` — вызывается при успешном выполнении асинхронной операции и подтверждает ее завершение. Значение, указанное в параметре `<Данные>`, будет доступно внутри функции, назначенной через первый параметр метода `then()`. Пример приведен в листинге 14.1.

Листинг 14.1. Использование метода `resolve()`

```
const myPromise = new Promise( function(resolve, reject) {
  setTimeout( function() {
    resolve('test');
  }, 500);
});
myPromise.then( function(value) {
  console.log(value);    // test
  console.log(myPromise); // Promise { 'test' }
});
console.log(myPromise);  // Promise { <pending> }
```

- `reject(<Объект исключения>)` — вызывается, напротив, при неуспешном выполнении асинхронной операции и отклоняет ее. Переданный ей объект исключения будет доступен внутри функции, назначенной через второй параметр метода `then()`, и внутри функции, назначенной методом `catch()`. Пример приведен в листинге 14.2.

Листинг 14.2. Использование метода `reject()`

```
const myPromise = new Promise( function(resolve, reject) {
  setTimeout( function() {
    reject( new Error('Описание ошибки') );
  }, 500);
});
myPromise.then(
  function(value) {          // Функция вызвана не будет
    console.log(value);
  },
  function(e) {
    console.log(e.message); // Описание ошибки
    console.log('test');
  });
myPromise.catch( function(e) {
  console.log(e.message);    // Описание ошибки
  console.log(myPromise);
  // Promise {
  //   <rejected> Error: Описание ошибки
}
```

```
//      at Timeout._onTimeout (C:\book\p1\index.js:5:15)
//      at listOnTimeout (internal/timers.js:549:17)
//      at processTimers (internal/timers.js:492:7)
// }
});
console.log(myPromise);      // Promise { <pending> }
```

Возвращаемое функцией значение игнорируется.

Объект `Promise` может находиться в следующих состояниях:

- ❑ `pending` — режим ожидания, когда асинхронная операция начала выполняться (начальное состояние);
- ❑ `fulfilled` — успешное выполнение операции;
- ❑ `rejected` — неуспешное выполнение (отклонение) операции.

Создать объект `Promise` также позволяют следующие статические методы:

- ❑ `resolve(<Данные>)` — возвращает объект `Promise` со статусом `fulfilled` (операция выполнена успешно). Значение, указанное в параметре `<Данные>`, будет доступно внутри функции, назначенной через первый параметр метода `then()`. Пример:

```
Promise.resolve('test').then( function(value) {
    console.log(value); // test
});
```

- ❑ `reject(<Объект исключения>)` — возвращает объект `Promise` со статусом `rejected` (операция отклонена). Объект исключения станет доступен внутри функции, назначенной через второй параметр метода `then()`, и внутри функции, назначенной методом `catch()`. Пример:

```
Promise.reject( new Error('Описание ошибки') )
    .then(
        function(value) {          // Функция вызвана не будет
            console.log(value);
        },
        function(e) {
            console.log(e.message); // Описание ошибки
        }
    );
```

14.1.2. Обработка изменения статуса

Обработать изменение статуса объекта `Promise` позволяют следующие методы:

- ❑ `then(<Функция1>[, <Функция2>])` — вызывает функцию `<Функция1>`, если операция успешно выполнена, или функцию `<Функция2>`, если операция отклонена. Метод возвращает объект `Promise`, содержащий результат, возвращаемый функцией `<Функция1>`, или `undefined`, если `<Функция1>` не возвращает результата. Пример:

```

Promise.resolve(10).then( function(value) {
  console.log(value);      // 10
  return value + 5;
}).then( function(value) {
  console.log(value);      // 15
}).then( function(value) {
  console.log(value);      // undefined
});

```

- ❑ `catch(<Функция>)` — вызывает заданную функцию, если операция отклонена. Внутри заданной функции через параметр доступен объект с описанием исключения. Пример:

```

const myPromise = new Promise( function(resolve, reject) {
  setTimeout( function() {
    reject( new Error('Описание ошибки') );
  }, 500);
});
myPromise.catch( function(e) {
  console.log(e.message);      // Описание ошибки
});

```

Функция, назначенная методом `catch()`, также будет вызвана при возникновении исключения внутри функции обратного вызова:

```

const myPromise = new Promise( function(resolve, reject) {
  throw new Error('Описание ошибки');
});
myPromise.catch( function(e) {
  console.log(e.message);      // Описание ошибки
});

```

Однако если исключение возникло внутри асинхронной функции, функция, назначенная методом `catch()`, вызвана не будет:

```

const myPromise = new Promise( function(resolve, reject) {
  setTimeout( function() {
    throw new Error('Описание ошибки');
  }, 500);
});
myPromise.catch( function(e) { // Функция вызвана не будет
  console.log(e);
});

```

Если исключение возникло после вызова `resolve()`, оно будет проигнорировано:

```

const myPromise = new Promise( function(resolve, reject) {
  resolve('test');
  throw new Error('Описание ошибки');
});
myPromise.then( function(value) {
  console.log(value);      // test
});

```

```
myPromise.catch( function(e) { // Функция вызвана не будет
  console.log(e);
});
```

- ❑ `finally(<Функция>)` — вызывает заданную функцию после выполнения функций, заданных в вызовах методов `then()` и `catch()`. Указываемая функция не должна ни принимать параметры, ни возвращать результат. Обычно в таких функциях выполняются какие-либо завершающие действия. Пример:

```
const myPromise = Promise.resolve('test');
myPromise.then( function(value) {
  console.log(value); // test
});
myPromise.finally( function() {
  console.log('Завершающие действия'); // Завершающие действия
});
```

Если операция была отклонена, метод `finally()` повторно генерирует исключение, приведшее к отмене операции:

```
const myPromise = new Promise( function(resolve, reject) {
  setTimeout( function() {
    reject( new Error('Ошибка!!!') );
  }, 500);
});
myPromise.catch( function(e) {
  console.log(e.message); // Ошибка!!!
});
myPromise.finally( function() {
  console.log('Завершающие действия'); // Завершающие действия
});
// Error: Ошибка!!!
// . . .
```

14.1.3. Массовая обработка объектов класса *Promise*

Класс `Promise` предоставляет ряд статических методов, служащих для массовой обработки сразу нескольких объектов этого класса. В качестве параметра эти методы принимают массивоподобный объект `<iterable>` (например, в самом простом случае массив), который должен хранить объекты класса `Promise`, подлежащие обработке. Вот эти статические методы:

- ❑ `all(<iterable>)` — возвращает объект класса `Promise`, который получит статус `fulfilled`, если все указанные объекты будут иметь статус `fulfilled`, или `rejected`, если хотя бы один из указанных объектов получит статус `rejected`.

В первом случае в функции, назначенной первым параметром в вызове метода `then()`, через параметр будет доступен массив со всеми значениями из выполнившихся объектов `Promise`:

```
const myPromise1 = new Promise( function(resolve, reject) {
  setTimeout( function() {
    resolve('test');
  }, 500);
});
const myPromise2 = Promise.resolve(10);
Promise.all([myPromise1, myPromise2]).then( function(values) {
  console.log(values);          // [ 'test', 10 ]
});
```

Во втором случае в функции, назначенной вторым параметром в вызове метода `then()` или в вызове метода `catch()`, через параметр будет доступен объект исключения, сгенерированного в отклоненном объекте (листинг 14.3).

Листинг 14.3. Метод `all()`

```
const myPromise1 = new Promise( function(resolve, reject) {
  setTimeout( function() {
    resolve('test');
  }, 500);
});
const myPromise2 = new Promise( function(resolve, reject) {
  setTimeout( function() {
    reject( new Error('Описание ошибки') );
  }, 300);
});
Promise.all([myPromise1, myPromise2]).then(
  function(values) {          // Функция не вызывается
    console.log(values);
  },
  function(e) {
    console.log(e.message);   // Описание ошибки
  });
```

- `any(<iterable>)` — возвращает объект класса `Promise`, который получит статус `fulfilled`, если хотя бы один из указанных объектов будет иметь статус `fulfilled`, или `rejected`, если все указанные объекты будут иметь статус `rejected`.

В первом случае в функции, назначенной первым параметром в вызове метода `then()`, через параметр будет доступно значение из первого успешно выполненного объекта `Promise` (листинг 14.4).

Листинг 14.4. Метод `any()` 1

```
const myPromise1 = new Promise( function(resolve, reject) {
  setTimeout( function() {
```

```
        resolve('test');
    }, 500);
});
const myPromise2 = Promise.reject( new Error('Ошибка!!!') );
Promise.any([myPromise1, myPromise2]).then( function(value) {
    console.log(value);          // test
});
```

Во втором случае метод `any()` самостоятельно генерирует исключение класса `AggregationError`, производное от класса `Error`. Класс `AggregationError` поддерживает свойство `errors`, хранящее массив с объектами исключений, которые были сгенерированы во всех отклоненных объектах. Объект исключения будет доступен через параметр в функции, назначенной вторым параметром в вызове метода `then()` или в вызове метода `catch()`. Пример приведен в листинге 14.5.

Листинг 14.5. Метод `any()` 2

```
const myPromise1 = new Promise( function(resolve, reject) {
    setTimeout( function() {
        reject( new Error('Опять ошибка!!!') );
    }, 500);
});
const myPromise2 = Promise.reject( new Error('Ошибка!!!') );
Promise.any([myPromise1, myPromise2]).then(
    function(values) {          // Функция не вызывается
        console.log(values);
    },
    function(es) {
        console.log(es.message);    // All promises were rejected
        for (const e of es.errors) {
            console.log(e.message); // Опять ошибка!!
        }
        // Ошибка!!!
    });
```

- ❑ `race(<iterable>)` — возвращает объект класса `Promise`, который получит статус `fulfilled` или `rejected`, в зависимости от статуса первого завершенного объекта (листинг 14.6).

Листинг 14.6. Метод `race()`

```
const myPromise1 = new Promise( function(resolve, reject) {
    setTimeout( function() {
        resolve('test');
    }, 500);
});
```



```

const myPromise2 = new Promise( function(resolve, reject) {
  setTimeout( function() {
    reject( new Error('Описание ошибки') );
  }, 300);
});
Promise.race( [myPromise1, myPromise2] ).then(
  function(value) {
    console.log(value);
  },
  function(e) {
    console.log(e.message); // Описание ошибки
  });

```

В этом примере первым был завершен объект `myPromise2` со статусом `rejected`. Поэтому возвращенный методом `race()` объект также получит статус `rejected`. Если мы увеличим время задержки с 300 до, скажем, 800, то первым будет завершен объект `myPromise1` со статусом `fulfilled`. Поэтому возвращенный методом `race()` объект также получит статус `fulfilled`, а значение `test` будет доступно через параметр `value`;

- `allSettled(<iterable>)` — возвращает объект класса `Promise`, который получит статус `fulfilled` после выполнения всех объектов, независимо от статуса. Значением объекта `Promise` будет массив результатов в виде простых объектов со свойствами `status` (значения: `'fulfilled'` или `'rejected'`), `value` (значение из выполнившегося объекта — при статусе `fulfilled`) или `reason` (объект исключения — при статусе `rejected`). Пример приведен в листинге 14.7.

Листинг 14.7. Метод `allSettled()`

```

const myPromise1 = new Promise( function(resolve, reject) {
  setTimeout( function() {
    resolve('test');
  }, 500);
});
const myPromise2 = new Promise( function(resolve, reject) {
  setTimeout( function() {
    reject( new Error('Описание ошибки') );
  }, 300);
});
Promise.allSettled( [myPromise1, myPromise2] ).then(
  function(values) {
    console.log(values);
  });
/* Результат:
[
  { status: 'fulfilled', value: 'test' },
  {
    status: 'rejected',

```

```
    reason: Error: Описание ошибки
      at Timeout._onTimeout (C:\book\pl\index.js:10:15)
      at listOnTimeout (internal/timers.js:549:17)
      at processTimers (internal/timers.js:492:7)
  }
] */
```

14.2. Ключевые слова *async* и *await*

Если перед объявлением функции указать ключевое слово `async`, то при возврате любого значения функция автоматически обернет это значение в объект класса `Promise` со статусом `fulfilled` и вернет этот объект:

```
async function func() {
  return 10;
}
func().then( function(value) {
  console.log(value);          // 10
});
```

Эквивалентный код:

```
function func() {
  return Promise.resolve(10);
}
func().then( function(value) {
  console.log(value);          // test
});
```

Внутри таких функций доступно ключевое слово `await`, которое позволяет приостановить выполнение функции, дождаться завершения асинхронной операции, выдающей объект класса `Promise`, и получить результат, завернутый в этот объект:

```
async function func() {
  const myPromise = new Promise( function(resolve, reject) {
    setTimeout( function() {
      resolve('test');
    }, 500);
  });
  let result = await myPromise;
  console.log(result); // test
}
func();
console.log('После вызова функции');
```

В консоли получим сообщения в следующем порядке:

```
После вызова функции
test
```

Как видим, код внутри функции выполняется асинхронно. Пока ожидается завершение операции, мы можем параллельно осуществлять какие-либо другие действия. При этом код внутри такой функции выполняется последовательно.

ПРИМЕЧАНИЕ

Ключевое слово `await` нельзя использовать внутри обычных функций или вне функций.

Если объект класса `Promise` завершается со статусом `rejected`, то будет сгенерировано исключение. Для его обработки можно использовать конструкцию `try...catch`. Пример приведен в листинге 14.8.

Листинг 14.8. Обработка исключения конструкцией `try...catch`

```
async function func() {
  const myPromise = new Promise( function(resolve, reject) {
    setTimeout( function() {
      reject( new Error('Описание ошибки') );
    }, 500);
  });
  try {
    let result = await myPromise;
    console.log(result);    // Инструкция выполнена не будет
  } catch (e) {
    console.log(e.message); // Описание ошибки
  }
}
func();
```

Учитывая, что функция возвращает объект `Promise`, мы можем обработать исключение с помощью метода `catch()` (листинг 14.9).

Листинг 14.9. Обработка исключений с помощью метода `catch()` 1

```
async function func() {
  const myPromise = new Promise( function(resolve, reject) {
    setTimeout( function() {
      reject( new Error('Описание ошибки') );
    }, 500);
  });
  let result = await myPromise;
  console.log(result);    // Инструкция выполнена не будет
}
func().catch( function(e) {
  console.log(e.message); // Описание ошибки
});
```

Можно также поступить следующим образом (листинг 14.10).

Листинг 14.10. Обработка исключений с помощью метода `catch()` 2

```
async function func() {
  const myPromise = new Promise( function(resolve, reject) {
    setTimeout( function() {
      reject( new Error('Описание ошибки') );
    }, 500);
  });
  let result = await myPromise.catch( function(e) {
    console.log(e.message); // Описание ошибки
  });
  console.log(result);      // undefined
}
func();
```

В этом случае переменная `result` будет содержать значение `undefined`.

Ключевое слово `async` также можно указывать перед стрелочными функциями (листинг 14.11).

Листинг 14.11. Асинхронная стрелочная функция

```
let func = async () => {
  const myPromise = new Promise( function(resolve, reject) {
    setTimeout( function() {
      resolve(10);
    }, 500);
  });
  let result = await myPromise;
  console.log(result);      // 10
}
func();
```

14.3. Цикл *for await...of*

Цикл `for await...of` используется для перебора массивоподобных объектов (например, массивов), хранящих объекты `Promise`. На каждой итерации он приостанавливает выполнение кода, ждет, когда очередной объект `Promise` не получит статус `fulfilled`, и заносит завернутое в объект значение в указанную переменную. Цикл имеет следующий формат:

```
for await (<Переменная> of <Объект>) {
  <Тело цикла>
}
```

Пример приведен в листинге 14.12.

Листинг 14.12. Использование цикла `for await...of`

```
function func(n, t) {
  return new Promise( function(resolve, reject) {
    setTimeout( function() {
      resolve(n);
    }, t);
  } );
}

(async function() {
  let arr = [ func(10, 500), func(20, 200), func(30, 5000) ];
  for await (const result of arr) {
    console.log(result);
  }
})();
console.log('Конец программы');
```

ОБРАТИТЕ ВНИМАНИЕ

Цикл `for await...of` можно использовать только внутри асинхронной функции.

Результат (последнее значение выводится со значительной задержкой):

```
Конец программы
10
20
30
```

Если очередной объект `Promise` получит статус `rejected`, будет сгенерировано исключение, причем для каждого такого объекта. При использовании инструкции `try...catch` будет обработано только первое исключение. Поэтому для каждого объекта `Promise` нужно назначить свой обработчик с помощью метода `catch()`. Внутри цикла при ошибке мы получим значение `undefined`. Пример приведен в листинге 14.13.

Листинг 14.13. Обработка исключений в цикле `for await...of`

```
function func(n, t) {
  return new Promise( function(resolve, reject) {
    setTimeout( function() {
      if (n == 20) resolve(n);
      else reject( new Error(`Ошибка. n = ${n}`) );
    }, t);
  } );
}

function errLog(e) {
  console.log(e.message);
}

(async function() {
  let arr = [ [10, 2000], [20, 200], [30, 5000] ];
```

```
let arrPromises = arr.map( e => func(e[0], e[1]).catch(errLog) );
for await (const result of arrPromises) {
  console.log(result);
}
})();
console.log('Конец программы');
```

Результат:

```
Конец программы
Ошибка. n = 10
undefined
20
Ошибка. n = 30
undefined
```

Цикл `for await...of` также позволяет перебирать обычные массивоподобные объекты:

```
(async function() {
  let arr = [ 10, 20, 30 ];
  for await (const result of arr) {
    console.log(result);
  }
})();
console.log('Конец программы');
```

Результат:

```
Конец программы
10
20
30
```

14.4. Асинхронные итераторы

Чтобы цикл `for await...of` мог перебирать пользовательские объекты, внутри объекта нужно реализовать поддержку асинхронных итераторов. Цикл `for await...of` сначала вызывает метод с именем `Symbol.asyncIterator`, а затем работает с возвращаемым им объектом, содержащим метод `next()`. Метод `next()` при каждом вызове должен возвращать объект `Promise`, содержащий объект с двумя свойствами: `done` и `value`. Перебор значений заканчивается, когда свойство `done` получит значение `true`. Пример приведен в листинге 14.14.

Листинг 14.14. Использование асинхронного итератора

```
'use strict';

let obj = {
  arr: [ 10, 20, 30 ],
```

```

[Symbol.asyncIterator]: function() {
  return {
    _index: 0,
    _arr: this.arr,
    next() {
      if (this._index < this._arr.length) {
        return Promise.resolve({
          done: false,
          value: this._arr[this._index++]
        });
      }
      else {
        return Promise.resolve({ done: true, value: undefined });
      }
    },
    toString() {
      return 'MyAsyncIterator {}';
    }
  };
}

};

(async function() {
  for await (const result of obj) {
    process.stdout.write(`${result} `);
  } // 10 20 30
})();

```

Создать асинхронный итератор и перебрать все значения вручную можно так:

```

(async function() {
  let it = obj[Symbol.asyncIterator]();
  console.log( it.toString() ); // MyAsyncIterator {}
  console.log( await it.next() ); // { done: false, value: 10 }
  console.log( await it.next() ); // { done: false, value: 20 }
  console.log( await it.next() ); // { done: false, value: 30 }
  console.log( await it.next() ); // { done: true, value: undefined }
})();

```

14.5. Асинхронные методы-генераторы

В классах можно определять асинхронные методы-генераторы, аналогичные обычным методам-генераторам (см. *разд. 11.8*). Имя такого метода должно предваряться символом звездочки (*). Также следует не забыть поставить перед именем ключевое слово `async`. Пример класса с асинхронным методом-генератором, выводящим четные числа в заданном диапазоне, приведен в листинге 14.15.

Листинг 14.15. Использование асинхронного метода-генератора

```
class Counter {
  async *evens(begin, end) {
    for (let i = begin; i <= end; i++) {
      if (i % 2 == 0) {
        yield i;
      }
    }
  }
}

const cnt = new Counter();
(async function() {
  for await (let n of cnt.evens(1, 5)) {
    console.log(n);
  }
})();
```

Результат в консоли:

```
2
4
```

Также можно определять приватные, статические и приватные статические асинхронные методы-генераторы:

```
class SomeClass {
  // Приватный асинхронный метод-генератор
  async *#method1() { . . . }
  // Публичный статический асинхронный метод-генератор
  static async *staticMethod2() { . . . }
  // Приватный статический асинхронный метод-генератор
  static async *#staticMethod3() { . . . }
}
```

14.6. Обработка событий

Событие — сигнал от программы о том, что в программе что-то произошло (например, завершилось выполнение запущенной ранее асинхронной операции). Каждое событие, которое может генерировать программа, должно иметь уникальное имя.

Обработка события — это реагирование на возникновение события заданным образом (например, получение результата завершившейся асинхронной операции). Обработка заданного события выполняется в особой функции-*обработчике*, которая заранее назначается этому событию.

События — другой механизм для отслеживания завершения асинхронных операций, применяемый наряду с объектами класса `Promise`.

Node.js обрабатывает события в особом цикле, который носит название *цикла обработки событий*. После возникновения какого-либо события интерпретатор запускает назначенные ему обработчики, дожидается их выполнения, после чего ждет возникновения очередного события. Если программа завершила свое выполнение, но какие-то события еще не обработаны, Node.js обрабатывает их и лишь после этого завершает работу программы.

Генерировать и обрабатывать события в Node.js позволяет класс `EventEmitter` из модуля `events`. Чтобы добавить механизм событий к пользовательскому классу, достаточно наследовать этот класс от класса `EventEmitter`. Пример приведен в листинге 14.16.

Листинг 14.16. Класс `EventEmitter`

```
const EventEmitter = require('events');
class MyClass extends EventEmitter {}
let obj = new MyClass();
// Назначение обработчика для события MyEvent
obj.on('MyEvent', () => {
  console.log('MyEvent');
});
// Генерирование события MyEvent
obj.emit('MyEvent');           // MyEvent
```

14.6.1. Генерирование событий

Сгенерировать событие (в результате чего будут выполнены все назначенные этому событию обработчики) позволяет метод `emit()` из класса `EventEmitter`. Формат метода:

```
emit(<Событие>[, ...<Значения через запятую>])
```

В первом параметре указывается имя генерируемого события в виде строки или `symbol`. Заданные следующими параметрами `<Значения>` будут переданы в обработчик события. Метод возвращает `true`, если у события существуют обработчики, и `false` — в противном случае. Пример генерирования и обработки событий `MyEvent1` и `MyEvent2` приведен в листинге 14.17.

Листинг 14.17. Генерирование и обработка событий

```
const EventEmitter = require('events');
let emitter = new EventEmitter();
emitter.on('MyEvent1', () => {
  console.log('MyEvent1');
});
emitter.on('MyEvent2', (a, b) => {
  console.log(`MyEvent2 a: ${a} b: ${b}`);
});
```

```
emitter.emit('MyEvent1');           // MyEvent1
emitter.emit('MyEvent2', 10, 6);    // MyEvent2 a: 10 b: 6
```

14.6.2. Назначение обработчиков событий

Каждому событию можно назначить произвольное число обработчиков, которые в этом случае организуются в подобие очереди в том порядке, в котором они были назначены. При генерировании события сначала выполняется первый обработчик из очереди, затем — второй и т. д.

Назначить обработчиков позволяют следующие методы из класса `EventEmitter`:

- `on(<Событие>, <Обработчик>)` и `addListener(<Событие>, <Обработчик>)` — добавляют обработчик в конец очереди. В первом параметре указывается имя обрабатываемого события в виде строки или `symbol`, во втором параметре — сама функция-обработчик. Методы возвращают ссылку на текущий объект `EventEmitter`. Пример назначения обработчиков приведен в листинге 14.18.

Листинг 14.18. Использование метода `on()`

```
const EventEmitter = require('events');
let emitter = new EventEmitter();
emitter.on('MyEvent', () => {
  console.log('MyEvent on()');
});
emitter.addListener('MyEvent', () => {
  console.log('MyEvent addListener()');
});
emitter.emit('MyEvent');
// MyEvent on()
// MyEvent addListener()
```

При использовании в качестве обработчика обычной функции внутри обработчика указатель `this` будет ссылаться на объект `EventEmitter`. Внутри стрелочных функций указатель `this` не ссылается на объект `EventEmitter`. Пример приведен в листинге 14.19.

Листинг 14.19. Использование метода `on()` со стрелочными функциями

```
const EventEmitter = require('events');
let emitter = new EventEmitter();
emitter.on('MyEvent', function() {
  console.log(this === emitter); // true
});
emitter.on('MyEvent', () => {
  console.log(this === emitter); // false
});
emitter.emit('MyEvent');
```

- ❑ `prependListener(<Событие>, <Обработчик>)` — вставляет обработчик в начало очереди. В первом параметре указывается имя события в виде строки или `symbol`, во втором параметре — функция-обработчик. Метод возвращает ссылку на текущий объект `EventEmitter`. Пример назначения обработчиков приведен в листинге 14.20.

Листинг 14.20 Использование метода `prependListener()`

```
const EventEmitter = require('events');
let emitter = new EventEmitter();
emitter.on('MyEvent', (n) => {
  console.log(`MyEvent on() ${n}`);
});
emitter.prependListener('MyEvent', (n) => {
  console.log(`MyEvent prependListener() ${n}`);
});
emitter.emit('MyEvent', 10);
// MyEvent prependListener() 10
// MyEvent on() 10
```

- ❑ `once(<Событие>, <Обработчик>)` — добавляет обработчик в конец очереди. Обработчик будет вызван только один раз, после чего — удален из очереди. В первом параметре указывается имя события в виде строки или `symbol`, во втором параметре — функция-обработчик. Метод возвращает ссылку на текущий объект `EventEmitter`. Пример назначения обработчика приведен в листинге 14.21.

Листинг 14.21. Использование метода `once()`

```
const EventEmitter = require('events');
let emitter = new EventEmitter();
emitter.once('MyEvent', (n, k) => {
  console.log(`MyEvent ${n}, ${k}`);
});
emitter.emit('MyEvent', 10, 3); // MyEvent 10 3
emitter.emit('MyEvent', 20, 5); // Нет вызова
```

- ❑ `prependOnceListener(<Событие>, <Обработчик>)` — вставляет обработчик в начало очереди. Обработчик будет вызван только один раз, после чего — удален из очереди. В первом параметре указывается имя события в виде строки или `symbol`, во втором параметре — функция-обработчик. Метод возвращает ссылку на текущий объект `EventEmitter`. Пример назначения обработчиков приведен в листинге 14.22.

Листинг 14.22. Использование метода `prependOnceListener()`

```
const EventEmitter = require('events');
let emitter = new EventEmitter();
```

```

emitter.once('MyEvent', (n, k) => {
  console.log(`MyEvent once() ${n}, ${k}`);
});
emitter.prependOnceListener('MyEvent', (n, k) => {
  console.log(`MyEvent prependOnceListener() ${n}, ${k}`);
});
emitter.emit('MyEvent', 10, 3);
// MyEvent prependOnceListener() 10, 3
// MyEvent once() 10, 3
emitter.emit('MyEvent', 20, 5); // Нет вызова

```

Получить массив с именами всех событий, которым были назначены обработчики, позволяет метод `eventNames()` из класса `EventEmitter` (листинг 14.23).

Листинг 14.23. Получение массива всех обрабатываемых событий

```

const EventEmitter = require('events');
let emitter = new EventEmitter();
emitter.on('MyEvent1', () => {
  console.log('MyEvent1');
});
emitter.on('MyEvent2', (a, b) => {
  console.log(`MyEvent2 a: ${a} b: ${b}`);
});
emitter.emit('MyEvent1'); // MyEvent1
emitter.emit('MyEvent2', 10, 6); // MyEvent2 a: 10 b: 6
console.log( emitter.eventNames() );
// [ 'MyEvent1', 'MyEvent2' ]

```

Узнать число зарегистрированных обработчиков события можно с помощью метода `listenerCount(<Событие>)`:

```

console.log( emitter.listenerCount('MyEvent1') ); // 1

```

Чтобы получить массив со ссылками на обработчики, следует воспользоваться методами `listeners(<Событие>)` или `rawListeners(<Событие>)`:

```

console.log( emitter.listeners('MyEvent1') ); // [ [Function] ]
console.log( emitter.rawListeners('MyEvent1') ); // [ [Function] ]

```

По умолчанию можно зарегистрировать 10 обработчиков одного события. Получить это значение позволяет метод `getMaxListeners()`. Задать другое значение позволяет метод `setMaxListeners(<Число>)`. Пример:

```

console.log( emitter.getMaxListeners() ); // 10
emitter.setMaxListeners(20);
console.log( emitter.getMaxListeners() ); // 20

```

Число обработчиков событий по умолчанию для всех объектов доступно через статическое свойство `defaultMaxListeners`:

```

console.log( EventEmitter.defaultMaxListeners ); // 10

```

14.6.3. Удаление обработчиков событий

Удалить обработчики событий позволяют следующие методы из класса `EventEmitter`:

- ❑ `off(<Событие>, <Обработчик>)` и `removeListener(<Событие>, <Обработчик>)` — удаляют один обработчик из конца очереди. В первом параметре указывается имя события в виде строки или `symbol`, во втором параметре — ссылка на функцию-обработчик. Методы возвращают ссылку на текущий объект `EventEmitter`. Пример удаления обработчиков приведен в листинге 14.24.

Листинг 14.24. Использование метода `off()`

```
const EventEmitter = require('events');
let emitter = new EventEmitter();
let listener1 = () => console.log('MyEvent listener1');
let listener2 = () => console.log('MyEvent listener2');
emitter.on('MyEvent', listener1);
emitter.on('MyEvent', listener2);
emitter.on('MyEvent', listener1);
emitter.emit('MyEvent');
// MyEvent listener1
// MyEvent listener2
// MyEvent listener1
emitter.off('MyEvent', listener1);
emitter.emit('MyEvent');
// MyEvent listener1
// MyEvent listener2
emitter.removeListener('MyEvent', listener2);
emitter.emit('MyEvent'); // MyEvent listener1
```

- ❑ `removeAllListeners([<Событие>])` — удаляет все обработчики. В качестве параметра указывается имя события в виде строки или `symbol`. Метод возвращает ссылку на текущий объект `EventEmitter`. Пример удаления обработчиков приведен в листинге 14.25.

Листинг 14.25. Использование метода `removeAllListeners()`

```
const EventEmitter = require('events');
let emitter = new EventEmitter();
let listener1 = () => console.log('listener1');
let listener2 = () => console.log('listener2');
emitter.on('MyEvent', listener1);
emitter.on('MyEvent', listener2);
emitter.emit('MyEvent');
// listener1
// listener2
```

```
emitter.removeAllListeners('MyEvent');  
emitter.emit('MyEvent'); // Нет вызова
```

Если имя события не задано, удаляются обработчики всех событий (листинг 14.26).

Листинг 14.26. Удаление всех обработчиков событий

```
const EventEmitter = require('events');  
let emitter = new EventEmitter();  
emitter.on('MyEvent1', () => console.log('listener1'));  
emitter.on('MyEvent2', () => console.log('listener2'));  
emitter.emit('MyEvent1'); // listener1  
emitter.emit('MyEvent2'); // listener2  
emitter.removeAllListeners();  
emitter.emit('MyEvent1'); // Нет вызова  
emitter.emit('MyEvent2'); // Нет вызова
```

14.6.4. Асинхронная обработка событий

По умолчанию обработчики вызываются синхронно в порядке добавления в очередь. Чтобы какой-либо обработчик события выполнялся асинхронно, нужно внутри него вызвать функцию `setImmediate()` (также можно использовать функцию `nextTick()` из объекта `process`). Они имеют одинаковый формат вызова:

```
setImmediate|process.nextTick(<Функция>[, <Значения через запятую>])
```

Функция `setImmediate()` предписывает Node.js выполнить заданную функцию после выполнения всех операций с более высоким приоритетом (операций файлового ввода-вывода, функций, указанных в вызовах `setInterval()` и `setTimeout()`, и др.). Указанные во втором и последующих параметрах значения будут переданы в заданную функцию при ее вызове.

Функция `nextTick()` дожидается, пока не закончится выполнение текущей операции, приостанавливает цикл обработки событий, выполняет заданную в ее вызове функцию, после чего вновь запускает цикл. Поэтому функции, заданные в вызове `nextTick()`, выполняются перед функциями, заданными в вызове `setImmediate()`.

Пример приведен в листинге 14.27.

Листинг 14.27. Применение функции `nextTick()`

```
const EventEmitter = require('events');  
let emitter = new EventEmitter();  
emitter.on('MyEvent', (n) => {  
  setImmediate((x) => {  
    console.log('Вызов обработчика. x =', x);  
  }, n);  
});
```

```
emitter.emit('MyEvent', 10);  
console.log('Конец программы');
```

Результат в консоли:

```
Конец программы  
Вызов обработчика. x = 10
```

Модуль `events` содержит функцию `on(<Объект emitter>, <Событие>)`, возвращающую асинхронный итератор, с помощью которого можно получить результаты выполнения обработчиков заданного события в виде массива (листинг 14.28).

Листинг 14.28. Применение функции `on()` из модуля `events`

```
const {on, EventEmitter} = require('events');  
(async function() {  
  let emitter = new EventEmitter();  
  setImmediate( () => {  
    emitter.emit('MyEvent', 10);  
    emitter.emit('MyEvent', 20);  
  });  
  process.nextTick( (v1, v2) => {  
    emitter.emit('MyEvent', v1);  
    emitter.emit('MyEvent', v2);  
  }, 30, 40);  
  for await (const data of on(emitter, 'MyEvent')) {  
    console.log(data);  
  }  
})();
```

Результат в консоли:

```
[ 30 ]  
[ 40 ]  
[ 10 ]  
[ 20 ]
```



ГЛАВА 15

Класс *Buffer*: массив байтов фиксированного размера

Строки в языке JavaScript всегда хранятся в кодировке UTF-16. Однако строки, получаемые извне (например, загружаемые из файлов), могут храниться в других кодировках и в процессе получения должны преобразовываться в UTF-16. А строки, предназначенные для вывода (скажем, для записи в файл), могут потребовать преобразования в какие-либо другие кодировки.

В процессе перекодирования строка сначала преобразуется в массив байтов, а потом — снова в строку, уже в другой кодировке. В Node.js такой массив байтов реализуется с помощью класса *Buffer*. Следует учитывать, что этот массив имеет фиксированный размер, изменить который после создания массива нельзя.

ПРИМЕЧАНИЕ

Помимо класса *Buffer* JavaScript поддерживает типизированные массивы: *Int8Array*, *Uint8Array*, *Int16Array*, *Uint16Array*, *Int32Array*, *Uint32Array* и др. Подробное описание этих классов можно найти в документации.

15.1. Создание массива байтов

Создать массив байтов позволяют следующие статические методы из класса *Buffer*:

- `alloc(<Размер>[, <Заполнитель>[, <Кодировка>]])` — создает массив, размер которого указан в первом параметре (задается в байтах). Если второй параметр не указан или имеет значение `undefined`, то массив заполняется нулями. Пример:

```
let buf = Buffer.alloc(5);
console.log(buf);          // <Buffer 00 00 00 00 00>
console.log(buf.length);  // 5
```

Если размер меньше нуля или больше значения константы `MAX_LENGTH`, то генерируется исключение `RangeError`. Выведем значение константы:

```
const buffer = require('buffer');
console.log(buffer.constants.MAX_LENGTH); // 2147483647
console.log(buffer.kMaxLength);           // 2147483647
```


Во втором параметре можно задать число, которое заполнит массив:

```
let buf = Buffer.alloc(5, 1);
console.log(buf);           // <Buffer 01 01 01 01 01>
```

Вместо числа можно указать другой объект класса `Buffer` — в результате будет создана копия указанного объекта:

```
let buf = Buffer.alloc(5, 1);
let buf2 = Buffer.alloc(5, buf);
console.log(buf2);          // <Buffer 01 01 01 01 01>
```

или объект класса `Uint8Array` — с аналогичным результатом:

```
let arr = new Uint8Array(
    [209, 130, 208, 181, 209, 129, 209, 130]);
let buf = Buffer.alloc(arr.length, arr);
console.log(buf);           // <Buffer d1 82 d0 b5 d1 81 d1 82>
console.log( buf.toString() ); // тест
```

или строку. В этом случае пригодится третий параметр, позволяющий задать кодировку. Доступны следующие кодировки: `'utf8'` (или `'utf-8'`, значение по умолчанию), `'utf16le'` (или `'utf-16le'`), `'latin1'`, `'base64'`, `'base64url'` и `'hex'`. Пример:

```
let buf = Buffer.alloc(4, 'test', 'latin1');
console.log(buf);           // <Buffer 74 65 73 74>
console.log( buf.toString('latin1') ); // test
buf = Buffer.alloc(8, 'тест', 'utf8');
console.log(buf);           // <Buffer d1 82 d0 b5 d1 81 d1 82>
console.log( buf.toString() ); // тест
buf = Buffer.alloc(8, 'd182d0b5d181d182', 'hex');
console.log(buf);           // <Buffer d1 82 d0 b5 d1 81 d1 82>
console.log( buf.toString() ); // тест
```

Если размер массива больше длины строки в байтах, строка будет повторяться:

```
let buf = Buffer.alloc(8, 'test', 'latin1');
console.log(buf);           // <Buffer 74 65 73 74 74 65 73 74>
console.log( buf.toString() ); // testtest
```

Рассчитать требуемый размер позволяет статический метод `byteLength()`:

```
let size = Buffer.byteLength('test', 'latin1');
let buf = Buffer.alloc(size, 'test', 'latin1');
console.log(buf);           // <Buffer 74 65 73 74>
console.log( buf.toString() ); // test
```

- ❑ `allocUnsafe(<Размер>)` — создает массив, размер которого указан в первом параметре (задается в байтах). Если размер меньше нуля или больше значения константы `MAX_LENGTH`, генерируется исключение `RangeError`. В отличие от метода `alloc()`, метод `allocUnsafe()` не заполняет элементы массива нулями, след-

ствие чего выполняется гораздо быстрее. Однако элементы массива будут содержать "мусорные" данные, оставшиеся от предыдущих операций. Пример:

```
let buf = Buffer.allocUnsafe(5);
console.log(buf); // <Buffer e0 da 65 ef 22>
```

Чтобы заполнить массив значениями, можно воспользоваться методом `fill()`:

```
let buf = Buffer.allocUnsafe(5);
buf.fill(0);
console.log(buf); // <Buffer 00 00 00 00 00>
```

Также можно указать в командной строке флаг `--zero-fill-buffers` — и все элементы массива получают значение 0 (правда, снизится производительность):

```
C:\book\p1>node index.js
<Buffer a0 d2 1e 6a 60>
```

```
C:\book\p1>node --zero-fill-buffers index.js
<Buffer 00 00 00 00 00>
```

- ❑ `allocUnsafeSlow(<Размер>)` — аналогичен `allocUnsafe()`, но небольшие (не более 4 Кбайт) массивы создает быстрее. Элементы массива будут содержать "мусорные" данные, оставшиеся от предыдущих операций. Чтобы заполнить массив значениями, можно воспользоваться методом `fill()`. Пример:

```
let buf = Buffer.allocUnsafeSlow(5);
console.log(buf); // <Buffer 31 fc 1f 02 94>
buf.fill(0);
console.log(buf); // <Buffer 00 00 00 00 00>
```

Также можно указать в командной строке флаг `--zero-fill-buffers` — и все элементы массива получают значение 0 (правда, снизится производительность):

```
C:\book\p1>node --zero-fill-buffers index.js
```

- ❑ `from(<Строка>[, <Кодировка>])` — создает массив байтов на основе строки. Во втором параметре можно задать кодировку. Доступны следующие кодировки: `'utf8'` (или `'utf-8'`, значение по умолчанию), `'utf16le'` (или `'utf-16le'`), `'latin1'`, `'base64'`, `'base64url'` и `'hex'`. Пример:

```
let buf = Buffer.from('test', 'latin1');
console.log(buf); // <Buffer 74 65 73 74>
console.log( buf.toString('latin1') ); // test
buf = Buffer.from('тест', 'utf8');
console.log(buf); // <Buffer d1 82 d0 b5 d1 81 d1 82>
console.log( buf.toString() ); // тест
buf = Buffer.from('d182d0b5d181d182', 'hex');
console.log(buf); // <Buffer d1 82 d0 b5 d1 81 d1 82>
console.log( buf.toString() ); // тест
```

- ❑ `from(<Массив байтов>)` — создает копию заданного массива байтов (объекта класса `Buffer`):

```
let buf = Buffer.from('test', 'latin1');
let buf2 = Buffer.from(buf);
buf2[0] = 0x6c;
buf2[1] = 0x69;
console.log( buf.toString('latin1') ); // test
console.log( buf2.toString('latin1') ); // list
```

Также можно указать объект класса Uint8Array:

```
let arr = new Uint8Array( [116, 101, 115, 116] );
let buf = Buffer.from(arr);
buf[0] = 0x6c;
buf[1] = 0x69;
console.log( buf ); // <Buffer 6c 69 73 74>
console.log( arr ); // Uint8Array(4) [ 116, 101, 115, 116 ]
```

- ❑ `from(<Массив>)` — создает объект класса `Buffer` на основе заданного обычного массива, содержащего числа от 0 до 255:

```
let arr = [116, 101, 115, 116];
let buf = Buffer.from(arr);
buf[0] = 0x6c;
buf[1] = 0x69;
console.log( buf ); // <Buffer 6c 69 73 74>
console.log( arr ); // [ 116, 101, 115, 116 ]
```

- ❑ `from(<ArrayBuffer>[, <Индекс>[, <Длина>]])` — создает представление буфера, реализуемого объектом класса `ArrayBuffer` или типизированным массивом (ссылка на буфер доступна через свойство `buffer` этих объектов). Копия данных не создается. Все изменения затронут исходный объект. Пример:

```
let arr = new Uint8Array( [116, 101, 115, 116] );
let buf = Buffer.from(arr.buffer);
buf[0] = 0x6c;
buf[1] = 0x69;
console.log( buf ); // <Buffer 6c 69 73 74>
                        // Данные изменились !!!
console.log( arr ); // Uint8Array(4) [ 108, 105, 115, 116 ]
```

Во втором параметре можно указать начальный индекс фрагмента, доступного через создаваемое представление, а в третьем — длину этого фрагмента в байтах:

```
let arr = new Uint8Array( [116, 101, 115, 116] );
let buf = Buffer.from(arr.buffer, 1, 2);
console.log( buf ); // <Buffer 65 73>
```

- ❑ `from(<Объект>[, <Кодировка>])` — создает массив байтов на основе значения, возвращаемого методом `[Symbol.toPrimitive]()` (должен возвращать строку) или `valueOf()` (должен возвращать строку или объект, например массив) объекта `<Объект>`. Во втором параметре можно указать кодировку. Если кодировка не задана, используется `'utf8'`. Пример приведен в листинге 15.1.

Листинг 15.1. Использование метода from()

```

let obj = {
  [Symbol.toPrimitive]() {
    return 'test';
  }
};
let buf = Buffer.from(obj, 'latin1');
console.log( buf ); // <Buffer 74 65 73 74>
obj = {
  valueOf() {
    return [0x74, 0x65, 0x73, 0x74];
  }
};
buf = Buffer.from(obj);
console.log( buf ); // <Buffer 74 65 73 74>
buf = Buffer.from( new String('тест') );
console.log( buf ); // <Buffer d1 82 d0 b5 d1 81 d1 82>
buf = Buffer.from( new String('тест'), 'utf8' );
console.log( buf ); // <Buffer d1 82 d0 b5 d1 81 d1 82>

```

- ❑ from(<JSON-объект>) — создает массив на основе объекта, возвращаемого методом toJSON():

```

let buf1 = Buffer.from( [1, 2, 3] );
let json = JSON.stringify(buf1);
console.log(json); // {"type":"Buffer","data":[1,2,3]}
let buf2 = Buffer.from( JSON.parse(json) );
console.log(buf2); // <Buffer 01 02 03>
let obj = buf1.toJSON();
console.log(obj); // { type: 'Buffer', data: [ 1, 2, 3 ] }
let buf3 = Buffer.from(obj);
console.log(buf3); // <Buffer 01 02 03>

```

Статический метод isBuffer(<Объект>) возвращает значение true, если <Объект> является объектом класса Buffer, и false — в противном случае:

```

let buf = Buffer.from('test');
console.log( Buffer.isBuffer(buf) ); // true
console.log( Buffer.isBuffer({}) ); // false
console.log( Buffer.isBuffer([]) ); // false

```

15.2. Определение размера массива

Получить размер массива (число хранящихся в нем байтов) позволяет свойство length:

```

let buf = Buffer.alloc(10);
console.log(buf.length); // 10

```

Статический метод `byteLength(<Объект>[, <Кодировка>])` из класса `Buffer` возвращает размер объекта в байтах. В первом параметре можно указать строку, объекты классов `Buffer`, `ArrayBuffer` и др. Если в первом параметре указана строка, то во втором параметре можно задать кодировку (значение по умолчанию: `'utf8'`). Пример:

```
console.log( Buffer.byteLength('test', 'latin1') ); // 4
let buf = Buffer.from('test', 'latin1');
console.log( Buffer.byteLength(buf) ); // 4
let arr = new Uint8Array( [116, 101, 115, 116] );
console.log( Buffer.byteLength(arr.buffer) ); // 4
```

15.3. Получение и изменение значения по индексу

Получить значение элемента массива можно, указав его индекс в квадратных скобках. Нумерация элементов массива начинается с 0. Пример:

```
let buf = Buffer.from( [116, 101, 115, 116] );
console.log( buf[0] ); // 116
console.log( buf[1] ); // 101
console.log( buf[2] ); // 115
console.log( buf[3] ); // 116
```

Если элемент с указанным индексом не существует, будет возвращено значение `undefined`:

```
console.log( buf[4] ); // undefined
```

Можно изменить значение существующего элемента:

```
buf[0] = 108;
buf[1] = 105;
console.log( buf[0] ); // 108
console.log( buf[1] ); // 105
```

15.4. Запись и чтение данных

С помощью метода `fill()` можно заполнить весь массив или только его диапазон одинаковым значением. Формат метода:

```
fill(<Значение>[, <Начало>[, <Конец>]][, <Кодировка>])
```

В первом параметре указывается значение-заполнитель в виде числа от 0 до 255, строки объекта `Buffer` или `Uint8Array`. Если указано число, оно заполнит весь массив. Пример:

```
let buf = Buffer.alloc(5);
buf.fill(1);
console.log(buf); // <Buffer 01 01 01 01 01>
```

В параметре <Начало> можно указать начальный индекс диапазона (значение по умолчанию: 0), а в параметре <Конец> — конечный индекс (значение по умолчанию: количество элементов массива). Метод `fill()` возвращает ссылку на массив, поэтому можно строить цепочки из вызовов. Пример:

```
let buf = Buffer.alloc(5).fill(1, 1, 4);
console.log(buf); // <Buffer 00 01 01 01 00>
```

Если вставляемое значение содержит несколько байтов и при этом размер массива больше, то значение будет повторяться:

```
let buf = Buffer.alloc(7);
buf.fill( Buffer.from( [1, 2] ) );
console.log(buf); // <Buffer 01 02 01 02 01 02 01>
buf.fill( new Uint8Array( [3, 4, 5] ) );
console.log(buf); // <Buffer 03 04 05 03 04 05 03>
```

Если в первом параметре указана строка, то в параметре <Кодировка> можно задать кодировку (значение по умолчанию: 'utf8'):

```
let buf = Buffer.alloc(7);
buf.fill('test');
console.log(buf); // <Buffer 74 65 73 74 74 65 73>
buf.fill(0).fill('test', 0, 4, 'latin1');
console.log(buf); // <Buffer 74 65 73 74 00 00 00>
```

Метод `write()` позволяет записать строку в массив. Формат метода:

```
write(<Строка>[, <Начало>[, <Длина>]][, <Кодировка>])
```

Метод возвращает число записанных байтов. Пример:

```
let buf = Buffer.alloc(10);
console.log( buf.write('test') ); // 4
console.log(buf); // <Buffer 74 65 73 74 00 00 00 00 00 00>
```

В параметре <Начало> указывается начальный индекс позиции вставки, в параметре <Длина> — максимальное число записываемых байтов, а в параметре <Кодировка> можно передать кодировку (значение по умолчанию: 'utf8'):

```
let buf = Buffer.alloc(10);
buf.write('test', 2, 3, 'latin1');
console.log(buf); // <Buffer 00 00 74 65 73 00 00 00 00 00>
```

Класс `Buffer` также содержит методы, позволяющие записать данные в различных форматах с указанным выравниванием: `writeInt8()`, `writeUInt8()`, `writeInt16BE()`, `writeInt16LE()`, `writeUInt16BE()`, `writeUInt16LE()`, `writeInt32BE()`, `writeInt32LE()`, `writeUInt32BE()`, `writeUInt32LE()`, `writeIntBE()`, `writeIntLE()`, `writeUIntBE()`, `writeUIntLE()`, `writeBigInt64BE()`, `writeBigInt64LE()`, `writeBigUInt64BE()`, `writeBigUInt64LE()`, `writeFloatBE()`, `writeFloatLE()`, `writeDoubleBE()` и `writeDoubleLE()`. Подробное описание этих методов можно найти в документации. Пример (в первом параметре указывается значение, а во втором — начальный индекс вставки):

```
let buf = Buffer.alloc(7);
buf.writeInt8(1);
console.log(buf);      // <Buffer 01 00 00 00 00 00 00>
buf.writeUInt8(2, 1);
console.log(buf);      // <Buffer 01 02 00 00 00 00 00>
buf.writeInt16BE(0x0304, 2);
console.log(buf);      // <Buffer 01 02 03 04 00 00 00>
buf.writeInt16LE(0x0506, 4);
console.log(buf);      // <Buffer 01 02 03 04 06 05 00>
```

Существуют также следующие методы, позволяющие прочитать значение из массива: `readInt8()`, `readUInt8()`, `readInt16BE()`, `readInt16LE()`, `readUInt16BE()`, `readUInt16LE()`, `readInt32BE()`, `readInt32LE()`, `readUInt32BE()`, `readUInt32LE()`, `readBigInt64BE()`, `readBigInt64LE()`, `readBigUInt64BE()`, `readBigUInt64LE()`, `readIntBE()`, `readIntLE()`, `readUIntBE()`, `readUIntLE()`, `readFloatBE()`, `readFloatLE()`, `readDoubleBE()` и `readDoubleLE()`. В первом параметре эти методы принимают смещение от начала массива и возвращают считанное значение. Подробное описание этих методов можно найти в документации.

В составе имен этих методов присутствуют тип данных и размер в битах. Если указанное в качестве параметра значение не входит в допустимый диапазон для типа данных, будет сгенерировано исключение. Далее перечислены диапазоны значений основных типов:

- ❑ `Int8` — целое число в диапазоне от -128 до 127 (занимает 1 байт);
- ❑ `UInt8` — целое число в диапазоне от 0 до 255 (занимает 1 байт);
- ❑ `Int16` — целое число в диапазоне от $-32\,768$ до $32\,767$ (2 байта);
- ❑ `UInt16` — целое число в диапазоне от 0 до $65\,535$ (2 байта);
- ❑ `Int32` — целое число в диапазоне от $-2\,147\,483\,648$ до $2\,147\,483\,647$ (4 байта);
- ❑ `UInt32` — целое число в диапазоне от 0 до $4\,294\,967\,295$ (4 байта);
- ❑ `BigInt64` — целое число в диапазоне от $-9\,223\,372\,036\,854\,775\,808$ до $9\,223\,372\,036\,854\,775\,807$ (8 байтов);
- ❑ `BigUInt64` — целое число в диапазоне от 0 до $18\,446\,744\,073\,709\,551\,615$ (8 байтов);
- ❑ `Float` — вещественное число (4 байта);
- ❑ `Double` — вещественное число двойной точности (8 байтов).

Пример записи и последующего считывания числа:

```
let buf = Buffer.alloc(8);
buf.writeBigUInt64BE(18446744073709551615n);
console.log( buf.readBigUInt64BE(0) );
// 18446744073709551615n
```

15.5. Создание копии массива

Следует помнить, что операция присваивания сохраняет в переменной копию ссылки на массив, а не копию самого массива. Например, при выполнении этого кода изменение `buf2` затронет `buf1`:

```
let buf1, buf2;
buf1 = Buffer.alloc(5);
buf2 = buf1;          // Присваивается ссылка на массив!!!
buf2[0] = 0x55;
console.log(buf1);    // <Buffer 55 00 00 00 00>
console.log(buf2);    // <Buffer 55 00 00 00 00>
```

Для создания полной копии массива можно воспользоваться статическим методом `from()` из класса `Buffer`:

```
let buf1, buf2;
buf1 = Buffer.alloc(5);
buf2 = Buffer.from(buf1);
buf2[0] = 0x55;
console.log(buf1);    // <Buffer 00 00 00 00 00>
console.log(buf2);    // <Buffer 55 00 00 00 00>
```

Скопировать фрагмент позволяет метод `slice(<Начало>[, <Конец>])` из класса `Uint8Array`. Если параметры не указаны, то создается полная копия. В первом параметре можно указать начальный индекс, а во втором — конечный индекс (не включая элемент с этим индексом). Если второй параметр не задан, копируется фрагмент, начиная с индекса `<Начало>` до конца массива. Пример приведен в листинге 15.2.

Листинг 15.2. Использование метода `slice()`

```
let buf1, buf2;
buf1 = Buffer.from( [0x01, 0x02, 0x03, 0x04, 0x05] );
buf2 = Uint8Array.prototype.slice.call(buf1);
buf2[0] = 0x55;
console.log(buf1);    // <Buffer 01 02 03 04 05>
console.log(buf2);    // <Buffer 55 02 03 04 05>
buf2 = Uint8Array.prototype.slice.call(buf1, 1);
console.log(buf2);    // <Buffer 02 03 04 05>
buf2 = Uint8Array.prototype.slice.call(buf1, 1, 3);
console.log(buf2);    // <Buffer 02 03>
```

Скопировать фрагмент из одного массива в другой позволяет метод `copy()`. Метод возвращает число записанных байтов. Формат метода:

```
buf1.copy(buf2[, <buf2 Начало>[, <buf1 Начало>[, <buf1 Конец>]]])
```

Метод копирует байты из массива `buf1` в массив `buf2` (объект класса `Buffer` или `Uint8Array`), начиная с индекса `<buf2 Начало>`. Если второй параметр не указан, ис-

пользуется значение 0. Третий параметр позволяет указать начальный индекс внутри массива `buf1` (значение по умолчанию: 0), а четвертый параметр — конечный индекс внутри массива `buf1` (не включая элемент с этим индексом; значение по умолчанию: размер массива `buf1`). Пример приведен в листинге 15.3.

Листинг 15.3. Применение метода `copy()`

```
let buf1, buf2;
buf1 = Buffer.from( [0x01, 0x02, 0x03, 0x04, 0x05] );
buf2 = Buffer.alloc(buf1.length);
console.log( buf1.copy(buf2) );           // 5
buf2[0] = 0x55;
console.log(buf1);      // <Buffer 01 02 03 04 05>
console.log(buf2);      // <Buffer 55 02 03 04 05>
buf2.fill(0);
console.log( buf1.copy(buf2, 2) );        // 3
console.log(buf2);      // <Buffer 00 00 01 02 03>
buf2.fill(0);
console.log( buf1.copy(buf2, 2, 1, 3) );  // 2
console.log(buf2);      // <Buffer 00 00 02 03 00>
buf2 = new Uint8Array( [0x00, 0x00, 0x00, 0x00, 0x00] );
console.log( buf1.copy(buf2) );           // 5
console.log(buf2);      // Uint8Array(5) [ 1, 2, 3, 4, 5 ]
```

15.6. Получение части массива

Метод `slice([<Начало>[, <Конец>]])` из класса `Buffer` возвращает срез массива, начиная от индекса `<Начало>` и заканчивая индексом `<Конец>`, но не включает элемент с этим индексом. Если первый параметр не указан, он будет иметь значение 0. Если второй параметр не указан, возвращаются все элементы до конца массива. Пример:

```
let buf1, buf2;
buf1 = Buffer.from( [0x01, 0x02, 0x03, 0x04, 0x05] );
buf2 = buf1.slice();
buf2[0] = 0x55;
console.log(buf1);      // <Buffer 55 02 03 04 05> !!!
console.log(buf2);      // <Buffer 55 02 03 04 05>
buf2 = buf1.slice(1, 3);
console.log(buf2);      // <Buffer 02 03>
```

Вместо метода `slice()` можно воспользоваться аналогичным методом `subarray([<Начало>[, <Конец>]])`:

```
let buf1, buf2;
buf1 = Buffer.from( [0x01, 0x02, 0x03, 0x04, 0x05] );
buf2 = buf1.subarray();
buf2[0] = 0x55;
console.log(buf1);      // <Buffer 55 02 03 04 05> !!!
```

```
console.log(buf2);    // <Buffer 55 02 03 04 05>
buf2 = buf1.subarray(1, 3);
console.log(buf2);    // <Buffer 02 03>
```

ВНИМАНИЕ!

Методы `slice()` и `subarray()` не копируют элементы, а лишь создают ссылку на ту же самую область памяти. Изменение элементов среза, созданного с помощью этих методов, приведет к изменению изначального массива.

15.7. Объединение массивов

Статический метод `concat(<Список массивов>[, <Размер>])` из класса `Buffer` возвращает массив, полученный в результате объединения массивов из числа указанных в списке. В первом параметре указывается массив объектов класса `Buffer` или `Uint8Array`. Во втором параметре можно задать размер нового массива. Если второй параметр не указан, размер нового массива будет равен сумме размеров массивов из списка. Если указано значение меньше этой суммы, результирующий массив будет иметь указанный размер. Пример:

```
let buf1 = Buffer.from( [0x01, 0x02, 0x03, 0x04, 0x05] );
let buf2 = Buffer.from( [0x06, 0x07, 0x08] );
let buf3 = Buffer.concat( [buf1, buf2] );
console.log(buf3); // <Buffer 01 02 03 04 05 06 07 08>
buf3 = Buffer.concat( [buf1, buf2], buf1.length + buf2.length);
console.log(buf3); // <Buffer 01 02 03 04 05 06 07 08>
buf3 = Buffer.concat( [buf1, buf2], 10);
console.log(buf3); // <Buffer 01 02 03 04 05 06 07 08 00 00>
buf3 = Buffer.concat( [buf1, buf2], 6);
console.log(buf3); // <Buffer 01 02 03 04 05 06>
```

Вместо объектов класса `Buffer` можно указать объекты класса `Uint8Array`:

```
let arr1 = new Uint8Array( [0x01, 0x02, 0x03, 0x04, 0x05] );
let arr2 = new Uint8Array( [0x06, 0x07, 0x08] );
let buf = Buffer.concat( [arr1, arr2] );
console.log(buf); // <Buffer 01 02 03 04 05 06 07 08>
```

15.8. Изменение порядка следования байтов

Изменить порядок следования байтов в массиве позволяют следующие методы из класса `Buffer`:

- `swap16()` — интерпретирует данные как 16-разрядные целые числа без знака и меняет порядок следования байтов. Метод возвращает ссылку на текущий массив. Пример:

```
let buf = Buffer.from( [0x01, 0x02, 0x03, 0x04] );
buf.swap16();
console.log(buf); // <Buffer 02 01 04 03>
```

Если размер массива не является кратным 2, генерируется исключение:

```
let buf = Buffer.from( [0x01, 0x02, 0x03] );
buf.swap16();
// RangeError [ERR_INVALID_BUFFER_SIZE]: Buffer size must be a
// multiple of 16-bits
```

- ❑ `swap32()` — интерпретирует данные как 32-разрядные целые числа без знака и меняет порядок следования байтов. Метод возвращает ссылку на текущий массив. Пример:

```
let buf = Buffer.from( [0x01, 0x02, 0x03, 0x04] );
buf.swap32();
console.log(buf); // <Buffer 04 03 02 01>
```

Если размер массива не является кратным 4, генерируется исключение:

```
let buf = Buffer.from( [0x01, 0x02, 0x03] );
buf.swap32();
// RangeError [ERR_INVALID_BUFFER_SIZE]: Buffer size must be a
// multiple of 32-bits
```

- ❑ `swap64()` — интерпретирует данные как 64-разрядные числа и меняет порядок следования байтов. Метод возвращает ссылку на текущий массив. Пример:

```
let buf = Buffer.from(
  [0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08]
);
buf.swap64();
console.log(buf); // <Buffer 08 07 06 05 04 03 02 01>
```

Если размер массива не является кратным 8, генерируется исключение:

```
let buf = Buffer.from( [0x01, 0x02, 0x03] );
buf.swap64();
// RangeError [ERR_INVALID_BUFFER_SIZE]: Buffer size must be a
// multiple of 64-bits
```

15.9. Перебор элементов массива

Для перебора массивов удобно использовать цикл `for`. Для примера умножим все элементы массива на 2:

```
let buf = Buffer.from( [1, 2, 3, 4] );
for (let i = 0; i < buf.length; i++) {
  buf[i] *= 2;
}
console.log(buf); // <Buffer 02 04 06 08>
```

Перебрать элементы массива позволяет также цикл `for...of`, который на каждой итерации присваивает указанной переменной копию значения очередного элемента массива:

```
let buf = Buffer.from( [1, 2, 3, 4] );
for (const value of buf) {
  console.log(value);
}
```

Также можно воспользоваться методом `forEach()`. Формат метода:

```
forEach(<Функция>[, <this>])
```

В первом параметре указывается ссылка на функцию, которая будет вызвана для каждого элемента массива. Функция принимает три параметра: через первый будет доступно значение очередного элемента, через второй — его индекс, а через третий — ссылка на массив. Значение, указанное в параметре `<this>`, будет доступно через указатель `this` внутри функции. Пример:

```
let buf = Buffer.from( [1, 2, 3, 4] );
buf.forEach( function(value, index, bf) {
  bf[index] = value * 2;
});
console.log(buf); // <Buffer 02 04 06 08>
```

15.10. Методы, возвращающие итераторы

Следующие методы возвращают особый объект — итератор:

□ `keys()` — возвращает итератор, содержащий индексы элементов массива:

```
let buf = Buffer.from( [5, 6, 7, 8] );
let iterator = buf.keys();
console.log(iterator); // Object [Array Iterator] {}
// Перебор всех элементов
for (const key of iterator) {
  process.stdout.write(`${key} `);
} // 0 1 2 3
console.log();
// Преобразование в массив
let arrKeys = [...buf.keys()];
console.log(arrKeys); // [ 0, 1, 2, 3 ]
```

□ `values()` — возвращает итератор, содержащий значения элементов массива:

```
let buf = Buffer.from( [5, 6, 7, 8] );
let iterator = buf.values();
// Перебор всех элементов
for (const value of iterator) {
  process.stdout.write(`${value} `);
} // 5 6 7 8
console.log();
// Преобразование в массив
let arrValues = [...buf.values()];
```

```
console.log(arrValues); // [ 5, 6, 7, 8 ]
arrValues = [...buf];
console.log(arrValues); // [ 5, 6, 7, 8 ]
```

- ❑ **entries()** — возвращает итератор, содержащий пары индекс/значение элементов массива:

```
let buf = Buffer.from( [5, 6, 7] );
let iterator = buf.entries();
// Перебор всех элементов
for (const [key, value] of iterator) {
  process.stdout.write(`${key} => ${value} `);
} // 0 => 5 1 => 6 2 => 7
console.log();
// Преобразование в массив
let arrEntries = [...buf.entries()];
console.log(arrEntries); // [ [ 0, 5 ], [ 1, 6 ], [ 2, 7 ] ]
```

15.11. Сравнение массивов

Сравнить два массива позволяет статический метод `compare(buf1, buf2)` из класса `Buffer`. В качестве параметров можно указать объекты классов `Buffer` и `Uint8Array`. Метод возвращает одно из значений:

- ❑ положительное число — если `buf1` больше `buf2`;
- ❑ отрицательное число — если `buf2` больше `buf1`;
- ❑ 0 — если объекты равны.

Пример:

```
let buf1 = Buffer.from( [1, 2, 3] );
let buf2 = Buffer.from( [1, 2, 3] );
let buf3 = Buffer.from( [1, 2, 2] );
let buf4 = Buffer.from( [1, 2, 4] );
console.log( Buffer.compare(buf1, buf2) ); // 0
console.log( Buffer.compare(buf1, buf3) ); // 1
console.log( Buffer.compare(buf1, buf4) ); // -1
let arr = [buf1, buf3];
arr.sort(Buffer.compare);
console.log(arr); // [ <Buffer 01 02 02>, <Buffer 01 02 03> ]
```

Сравнить два массива или их фрагменты позволяет метод `compare()`. Формат метода:

```
buf1.compare(buf2[, <buf2 Начало>[, <buf2 Конец>[,
  <buf1 Начало>[, <buf1 Конец>]]]])
```

В качестве первого параметра можно указать объекты классов `Buffer` и `Uint8Array`. Метод возвращает одно из значений:

- положительное число — если buf1 больше buf2;
- отрицательное число — если buf2 больше buf1;
- 0 — если объекты или их фрагменты равны.

Пример:

```
let buf1 = Buffer.from( [1, 2, 3] );
let buf2 = Buffer.from( [1, 2, 3] );
let buf3 = Buffer.from( [1, 2, 2] );
let buf4 = Buffer.from( [1, 2, 4] );
console.log( buf1.compare(buf2) ); // 0
console.log( buf1.compare(buf3) ); // 1
console.log( buf1.compare(buf4) ); // -1
```

Метод сравнивает байты массивов buf1 и buf2, начиная с индекса <buf2 Начало>. Если второй параметр не указан, используется значение 0. Третий параметр задает конечный индекс внутри массива buf2 (не включая элемент с этим индексом; значение по умолчанию: размер массива buf2). Четвертый параметр позволяет указать начальный индекс внутри массива buf1 (значение по умолчанию: 0), а пятый — конечный индекс внутри массива buf1 (не включая элемент с этим индексом; значение по умолчанию: размер массива buf1). Пример:

```
let buf1 = Buffer.from( [1, 2, 3] );
let buf2 = Buffer.from( [1, 2, 2] );
let buf3 = Buffer.from( [1, 2, 4] );
console.log( buf1.compare(buf2, 0, buf2.length,
                                0, buf1.length) ); // 1
console.log( buf1.compare(buf2, 0, 2, 0, 2) ); // 0
console.log( buf1.compare(buf3, 0, buf3.length,
                                0, buf1.length) ); // -1
console.log( buf1.compare(buf3, 0, 2, 0, 2) ); // 0
```

Метод equals(<Объект>) сравнивает текущий объект с объектом, указанным в качестве параметра (объект класса Buffer или Uint8Array), и возвращает значение true, если объекты равны, или false — в противном случае:

```
let buf1 = Buffer.from( [1, 2, 3] );
let buf2 = Buffer.from( [1, 2, 3] );
let buf3 = Buffer.from( [1, 2, 4] );
console.log( buf1.equals(buf2) ); // true
console.log( buf1.equals(buf3) ); // false
```

15.12. Проверка наличия значения в массиве

Выполнить поиск значения в массиве позволяют следующие методы:

- indexOf(<Значение>[, <Индекс>][, <Кодировка>]) — ищет первое вхождение заданного значения в текущем массиве. В первом параметре можно указать число в диапазоне от 0 до 255. Метод возвращает индекс первого вхождения или значение -1, если значение не найдено. Во втором параметре можно указать индекс,

с которого начнется поиск. Если второй параметр не указан, поиск начинается с начала массива. Пример:

```
let buf = Buffer.from( [1, 2, 3] );
console.log( buf.indexOf(2) );           // 1
console.log( buf.indexOf(2, 2) );        // -1
console.log( buf.indexOf(5) );           // -1
```

Если в первом параметре указана строка, то в параметре <Кодировка> можно указать кодировку строки (значение по умолчанию: 'utf8'):

```
let buf = Buffer.from('тест текст');
console.log( buf.indexOf('ст', 'utf8') ); // 4
console.log( buf.indexOf('ст', 8, 'utf8') ); // 15
console.log( buf.indexOf('ет', 0, 'utf8') ); // -1
```

В первом параметре также можно указать объект класса Buffer или Uint8Array:

```
let buf1 = Buffer.from('тест тест');
let buf2 = Buffer.from('тест');
console.log( buf1.indexOf(buf2) );        // 0
console.log( buf1.indexOf(buf2, 8) );     // 9
```

- `lastIndexOf(<Значение>[, <Индекс>][, <Кодировка>])` — аналогичен методу `indexOf()`, только ищет последнее вхождение значения в массиве:

```
let buf = Buffer.from( [1, 2, 3, 2, 4] );
console.log( buf.lastIndexOf(2) );        // 3
console.log( buf.lastIndexOf(2, 2) );     // 1
console.log( buf.lastIndexOf(5) );        // -1
let buf2 = Buffer.from('тест тест');
let buf3 = Buffer.from('тест');
console.log( buf2.lastIndexOf(buf3) );    // 9
console.log( buf2.lastIndexOf(buf3, 8) ); // 0
console.log( buf2.lastIndexOf('тест', 'utf8') ); // 9
console.log( buf2.lastIndexOf('тест', 8, 'utf8') ); // 0
```

- `includes(<Значение>[, <Индекс>][, <Кодировка>])` — возвращает значение `true`, если заданное значение присутствует в текущем массиве, и `false` — в противном случае. Во втором параметре можно указать индекс элемента, с которого начнется поиск. Пример:

```
let buf = Buffer.from( [1, 2, 3] );
console.log( buf.includes(2) );           // true
console.log( buf.includes(2, 2) );        // false
console.log( buf.includes(5) );           // false
let buf2 = Buffer.from('тест');
let buf3 = Buffer.from('тест');
console.log( buf2.includes(buf3) );       // true
console.log( buf2.includes(buf3, 2) );    // false
console.log( buf2.includes('тест', 'utf8') ); // true
console.log( buf2.includes('тест', 2, 'utf8') ); // false
```

15.13. Преобразование массива в строку или в другой объект

Преобразовать массив в строку позволяет метод `toString([<Кодировка>[, <Начало>[, <Конец>]])`. В первом параметре можно указать кодировку строк внутри массива (значение по умолчанию: `'utf8'`). Второй параметр задает начальный индекс внутри массива, а третий параметр — конечный индекс (элемент с этим индексом не входит в диапазон). Пример:

```
let buf = Buffer.from('тест текст', 'utf8');
console.log( buf.toString() );                // тест текст
console.log( buf.toString('utf8') );          // тест текст
console.log( buf.toString('utf8', 9, buf.length) ); // текст
```

Максимальная длина строки в кодировке UTF-16 доступна через константу `MAX_STRING_LENGTH`:

```
const buffer = require('buffer');
console.log(buffer.constants.MAX_STRING_LENGTH); // 1073741799
```

Метод `toJSON()` возвращает объект для преобразования в формат JSON:

```
let buf = Buffer.from( [1, 2, 3] );
console.log( buf.toJSON() ); // { type: 'Buffer', data: [ 1, 2, 3 ] }
```

Возвращаемое этим методом значение используется методом `stringify()` для создания строки в формате JSON:

```
let buf = Buffer.from( [1, 2, 3] );
console.log( JSON.stringify(buf) ); // {"type":"Buffer","data":[1,2,3]}
```

В дальнейшем можно преобразовать строку в формате JSON в объект и передать его статическому методу `from()` из класса `Buffer`:

```
let json = '{"type":"Buffer","data":[1,2,3]}';
let buf = Buffer.from( JSON.parse(json) );
console.log(buf); // <Buffer 01 02 03>
```

Для преобразования объекта в обычный массив достаточно воспользоваться оператором `SPREAD`:

```
let buf = Buffer.from( [1, 2, 3] );
let arr = [...buf];
console.log( arr );                // [ 1, 2, 3 ]
console.log( [...buf.values()] );  // [ 1, 2, 3 ]
console.log( [...buf.keys()] );    // [ 0, 1, 2 ]
console.log( [...buf.entries()] ); // [ [ 0, 1 ], [ 1, 2 ], [ 2, 3 ] ]
```

Для преобразования объекта в типизированный массив достаточно передать объект конструктору соответствующего класса:

```
let buf = Buffer.from( [1, 2, 3] );
let arr = new Uint8Array(buf);
```



```
console.log(arr); // Uint8Array(3) [ 1, 2, 3 ]
arr[0] = 55;
console.log(arr); // Uint8Array(3) [ 55, 2, 3 ]
console.log(buf); // <Buffer 01 02 03>
```

Если нужно, чтобы типизированный массив ссылался на ту же самую область памяти, что и исходный объект, следует воспользоваться свойствами `buffer` и `byteOffset`:

```
let buf = Buffer.from( [1, 2, 3] );
let arr = new Uint8Array(buf.buffer, buf.byteOffset,
                        buf.length / Uint8Array.BYTES_PER_ELEMENT);
console.log(arr); // Uint8Array(3) [ 1, 2, 3 ]
arr[0] = 55;
console.log(arr); // Uint8Array(3) [ 55, 2, 3 ]
console.log(buf); // <Buffer 37 02 03>
```

15.14. Преобразование кодировок

Node.js поддерживает ограниченный набор кодировок: `'utf8'` (или `'utf-8'`, кодировка по умолчанию), `'utf16le'` (или `'utf-16le'`), `'latin1'`, `'base64'`, `'base64url'` и `'hex'`, а также устаревшие `'ascii'`, `'binary'` и `'ucs2'` (или `'ucs-2'`). Эти кодировки можно указать при создании массива и при преобразовании массива в строку, например:

```
let buf = Buffer.from('тест', 'utf16le');
console.log( buf.toString('utf16le') ); // тест
```

Для преобразования кодировки после создания массива следует воспользоваться функцией `transcode(<Массив>, <Текущая кодировка>, <Новая кодировка>)` из модуля `buffer`. Доступны кодировки: `'utf8'` (или `'utf-8'`), `'utf16le'` (или `'utf-16le'`) и `'latin1'`, а также устаревшие `'ascii'`, `'binary'` и `'ucs2'` (или `'ucs-2'`). Функция возвращает новый массив. Пример:

```
const buffer = require('buffer');
let buf = Buffer.from('тест', 'utf16le');
let buf2 = buffer.transcode(buf, 'utf16le', 'utf-8');
console.log( buf.toString('utf-16le') ); // тест
console.log( buf2.toString('utf8') ); // тест
```

Если указана неподдерживаемая кодировка, будет сгенерировано исключение:

```
let buf = Buffer.from('тест', 'windows-1251');
// TypeError [ERR_UNKNOWN_ENCODING]: Unknown encoding: windows-1251
```

Проверить, поддерживается ли заданная кодировка, позволяет статический метод `isEncoding(<Кодировка>)` из класса `Buffer`. Метод возвращает значение `true`, если кодировка поддерживается, и `false` — в противном случае. Пример:

```
console.log( Buffer.isEncoding('utf8') );    // true
console.log( Buffer.isEncoding('utf-8') );    // true
console.log( Buffer.isEncoding('utf16le') );  // true
console.log( Buffer.isEncoding('latin1') );    // true
console.log( Buffer.isEncoding('base64') );    // true
console.log( Buffer.isEncoding('hex') );       // true
// Устаревшие кодировки
console.log( Buffer.isEncoding('ascii') );     // true
console.log( Buffer.isEncoding('binary') );    // true
console.log( Buffer.isEncoding('ucs2') );      // true
console.log( Buffer.isEncoding('ucs-2') );     // true
```

Русские кодировки не поддерживаются:

```
console.log( Buffer.isEncoding('windows-1251') ); // false
console.log( Buffer.isEncoding('windows-866') );  // false
console.log( Buffer.isEncoding('cp1251') );       // false
console.log( Buffer.isEncoding('cp866') );        // false
```

Чтобы расширить перечень доступных кодировок, нужно установить пакет `iconv-lite`. Для этого в командной строке выполняем команду:

```
npm install iconv-lite
```

В файле `package.json` должна появиться строка:

```
"iconv-lite": "^0.6.2"
```

Проверим наличие поддержки русских кодировок с помощью функции `encodingExists(<Кодировка>)` из модуля `iconv-lite`:

```
const iconv = require('iconv-lite');
console.log( iconv.encodingExists('windows-1251') ); // true
console.log( iconv.encodingExists('windows1251') ); // true
console.log( iconv.encodingExists('win1251') );     // true
console.log( iconv.encodingExists('windows-866') ); // false
console.log( iconv.encodingExists('cp1251') );      // true
console.log( iconv.encodingExists('cp866') );       // true
console.log( iconv.encodingExists('koi8-r') );      // true
```

Как видно из результата, все кодировки стали доступны, кроме `'windows-866'`, вместо которой нужно использовать `'cp866'`.

Создать массив байтов в нужной кодировке на основе заданной строки позволяет функция `encode(<Строка>, <Кодировка>[, <Опции>])` из модуля `iconv-lite`. Чтобы преобразовать массив байтов `Buffer` в строку, следует воспользоваться функцией `decode(<Buffer>, <Кодировка>[, <Опции>])` из того же модуля. Пример:

```
const iconv = require('iconv-lite');
let buf = iconv.encode('тест', 'win1251');
console.log(buf); // <Buffer f2 e5 f1 f2>
let str = iconv.decode(buf, 'win1251');
console.log(str); // тест
```



ГЛАВА 16

Чтение и запись файлов

Почти все функции, выполняющие чтение и запись файлов, расположены во встроенном модуле `fs`. Не забудьте при использовании модуля CommonJS в начало программы добавить следующую инструкцию:

```
const fs = require('fs');
```

Внутри модуля ECMAScript инструкция выглядит так:

```
import * as fs from 'fs';
```

16.1. Указание пути к файлу или каталогу

В каждом модуле типа CommonJS доступны следующие две переменные:

- `__dirname` — абсолютный путь к каталогу с модулем;
- `__filename` — абсолютный путь к файлу модуля.

Выведем значения переменных внутри файла `C:\book\p1\index.js`:

```
console.log(__dirname); // C:\book\p1
console.log(__filename); // C:\book\p1\index.js
```

В модуле ECMAScript получить URL с полным путем к текущему файлу позволяет свойство `import.meta.url`. Выведем значение свойства внутри файла `C:\book\p1\test.mjs`, а также создадим константы `__dirname` и `__filename`:

```
import { dirname } from 'path';
import { fileURLToPath } from 'url';
console.log(import.meta.url); // file:///C:/book/p1/test.mjs
const __filename = fileURLToPath(import.meta.url);
const __dirname = dirname(__filename);
console.log(__dirname); // C:\book\p1
console.log(__filename); // C:\book\p1\test.mjs
```

Для работы с путями в Node.js предназначен модуль `path`. Инструкция импорта внутри модуля CommonJS:

```
const path = require('path');
```

Инструкция импорта внутри модуля ECMAScript:

```
import * as path from 'path';
```

По умолчанию работа с путями зависит от текущей операционной системы, поэтому при запуске программы в Windows пути следует указывать в стиле Windows. Получить информацию о текущей операционной системе позволяет встроенный модуль `os`. Пример:

```
const os = require('os');
console.log( os.platform() );           // win32
console.log( os.version() );           // Windows 10 Pro
console.log( os.type() );              // Windows_NT
```

Чтобы не зависеть от текущей операционной системы, можно воспользоваться свойствами `win32` и `posix`:

```
const path = require('path');
console.log( path.basename(__filename) );           // index.js
console.log( path.win32.basename(__filename) );    // index.js
console.log( path.posix.basename(__filename) );    // C:\book\p1\index.js
```

Как видно из последней инструкции, если передать путь, не соответствующий операционной системе, то результат будет неправильным. Если мы передадим правильный путь, все будет нормально, например:

```
const p = '/book/p1/index.js';
console.log( path.posix.basename(p) ); // index.js
```

Путь может быть *абсолютным* или *относительным*. При указании абсолютного пути в Windows следует учитывать, что слеш является специальным символом. По этой причине его необходимо удваивать:

```
console.log('C:\\temp\\new\\file.txt'); // C:\temp\new\file.txt
```

Если вместо двух слешей поставить один, в пути будут присутствовать сразу три специальных символа: `\t`, `\n` и `\f`. После преобразования специальных символов путь будет выглядеть следующим образом:

```
C:<Табуляция>emp<Перевод строки>ew<Перевод формата>ile.txt
```

Для указания пути можно также использовать неформатированные строки (см. *разд. 7.4*). В этом случае экранировать слеш не нужно. Пример:

```
console.log(String.raw`C:\temp\new\file.txt`); // C:\temp\new\file.txt
```

Вместо абсолютного пути к файлу можно указать относительный, отсчитываемый от *текущего рабочего каталога*. Обратите внимание на то, что текущий рабочий каталог не всегда совпадает с каталогом, в котором находится исполняемый файл. Если файл запускается из командной строки, то текущим рабочим каталогом будет каталог, из которого запускается файл. Получить путь к текущему рабочему каталогу позволяет функция `cwd()` из модуля `process`, а также функция `resolve()` из модуля `path`, если она вызвана без параметра. Пример:

```
const path = require('path');
console.log( 'cwd: ' + process.cwd() );
console.log( 'cwd: ' + path.resolve() );
console.log( path.resolve('index.js') );
```

Запустим программу из командной строки разными способами:

```
C:\book\p1>node index.js
cwd: C:\book\p1
cwd: C:\book\p1
C:\book\p1\index.js
```

```
C:\book\p1>cd ..
```

```
C:\book>node p1\index.js
cwd: C:\book
cwd: C:\book
C:\book\index.js
```

Чтобы не зависеть от текущего рабочего каталога, следует воспользоваться переменной `__dirname` и функцией `resolve()` или `join()`:

```
const path = require('path');
console.log( 'cwd: ' + path.resolve() );
console.log( path.resolve(__dirname, 'index.js') );
console.log( path.join(__dirname, 'index.js') );
```

Запустим программу из командной строки разными способами:

```
C:\book\p1>node index.js
cwd: C:\book\p1
C:\book\p1\index.js
C:\book\p1\index.js
```

```
C:\book\p1>cd ..
```

```
C:\book>node p1\index.js
cwd: C:\book
C:\book\p1\index.js
C:\book\p1\index.js
```

```
C:\book>cd ..
```

```
C:\>node book\p1\index.js
cwd: C:\
C:\book\p1\index.js
C:\book\p1\index.js
```

Возможны следующие варианты указания относительного пути в Windows:

- если открываемый файл находится в текущем рабочем каталоге, то можно указать только имя файла или перед именем файла добавить одну точку и прямой или обратный слеш:

```
const path = require('path');
console.log( path.resolve(__dirname, 'file.txt') );
// C:\book\pl\file.txt
console.log( path.resolve(__dirname, './file.txt') );
// C:\book\pl\file.txt
console.log( path.resolve(__dirname, '..\file.txt') );
// C:\book\pl\file.txt
```

- ❑ если открываемый файл расположен во вложенном каталоге, то перед именем файла перечисляются имена вложенных каталогов через прямой или обратный слеш:

```
console.log( path.resolve(__dirname, 'f1\\file.txt') );
// C:\book\pl\f1\file.txt
console.log( path.resolve(__dirname, 'f1/file.txt') );
// C:\book\pl\f1\file.txt
console.log( path.resolve(__dirname, 'f1\\f2\\file.txt') );
// C:\book\pl\f1\f2\file.txt
console.log( path.resolve(__dirname, './f1/f2/file.txt') );
// C:\book\pl\f1\f2\file.txt
```

- ❑ если каталог с файлом расположен выше уровнем, то перед именем файла указываются две точки и прямой или обратный слеш:

```
console.log( path.resolve(__dirname, '..\\file.txt') );
// C:\book\file.txt
console.log( path.resolve(__dirname, '../file.txt') );
// C:\book\file.txt
console.log( path.resolve(__dirname, '..\\..\file.txt') );
// C:\file.txt
console.log( path.resolve(__dirname, '../../file.txt') );
// C:\file.txt
```

- ❑ если в начале пути расположен слеш, то путь отсчитывается от корня текущего диска. В этом случае местоположение текущего рабочего каталога не имеет значения. Пример:

```
console.log( path.resolve(__dirname, '\\book\\test\\file.txt') );
// C:\book\test\file.txt
console.log( path.resolve(__dirname, '/book/test/file.txt') );
// C:\book\test\file.txt
```

- ❑ если открываемый файл находится на другом диске, обозначение диска следует указать в начале пути и поставить после него двоеточие:

```
console.log( path.resolve(__dirname, 'd:\\work\\pl\\file.txt') );
// d:\work\pl\file.txt
console.log( path.resolve(__dirname, 'd:/work/pl/file.txt') );
// d:\work\pl\file.txt
```

16.2. Модуль *path*: преобразование путей

Модуль `path` содержит следующие свойства:

- `win32` — предоставляет доступ к методам, обрабатывающим пути в стиле Windows:

```
const path = require('path');
const p = 'C:\\book\\p1\\index.js';
console.log( path.win32.basename(p) ); // index.js
```

- `posix` — предоставляет доступ к методам, обрабатывающим пути в стиле POSIX:

```
const p2 = '/book/p1/index.js';
console.log( path.posix.basename(p2) ); // index.js
```

- `delimiter` — символ-разделитель путей, используемый в текущей операционной системе:

```
console.log(path.delimiter); // B Windows: ;
console.log(path.win32.delimiter); // ;
console.log(path.posix.delimiter); // :
```

- `sep` — символ-разделитель фрагментов в пути, используемый в текущей операционной системе:

```
console.log(path.sep); // B Windows: \
console.log(path.win32.sep); // \
console.log(path.posix.sep); // /
```

Модуль `path` содержит следующие функции:

- `isAbsolute(<Путь>)` — возвращает значение `true`, если путь абсолютный, и `false` — если относительный:

```
const path = require('path');
let p = 'C:\\book\\p1';
console.log( path.isAbsolute(p) ); // true
console.log( path.win32.isAbsolute(p) ); // true
console.log( path.posix.isAbsolute(p) ); // false
p = '/book/p1';
console.log( path.win32.isAbsolute(p) ); // true
console.log( path.posix.isAbsolute(p) ); // true
p = 'book\\p1';
console.log( path.isAbsolute(p) ); // false
console.log( path.win32.isAbsolute(p) ); // false
console.log( path.posix.isAbsolute(p) ); // false
```

- `normalize(<Путь>)` — возвращает нормализованный путь:

```
let p = 'C:\\book\\..\\file.txt';
console.log( path.normalize(p) ); // C:\\file.txt
console.log( path.win32.normalize(p) ); // C:\\file.txt
p = '/book/../file.txt';
```

```
console.log( path.posix.normalize(p) ); // /file.txt
p = 'C:/book\../file.txt';
console.log( path.win32.normalize(p) ); // C:\file.txt
p = '';
console.log( path.normalize(p) );      // .
```

- ❑ `join(<Сегменты через запятую>)` — возвращает путь, составленный из заданных сегментов и нормализованный:

```
console.log( path.join('C:', 'book', 'file.txt') );
// C:\book\file.txt
let p = ['book', 'pl', 'file.txt'];
console.log( path.join(...p) );           // book\pl\file.txt
console.log( path.win32.join(...p) );     // book\pl\file.txt
console.log( path.posix.join(...p) );     // book/pl/file.txt
p = ['book', 'pl', '..', 'file.txt'];
console.log( path.join(...p) );           // book\file.txt
console.log( path.join('') );             // .
console.log( path.join(__dirname, 'file.txt') );
// C:\book\pl\file.txt
```

- ❑ `resolve(<Сегменты через запятую>)` — возвращает абсолютный путь, составленный из заданных сегментов и нормализованный. Если после объединения сегментов не удалось получить абсолютный путь, то путь достраивается относительно текущего рабочего каталога. Пример:

```
console.log( path.resolve('C:\\book', 'pl', 'file.txt') );
// C:\book\pl\file.txt
console.log( path.resolve(__dirname, 'file.txt') );
// C:\book\pl\file.txt
console.log( path.resolve(__dirname, '../file.txt') );
// C:\book\file.txt
let p = ['/book', '/pl', 'file.txt'];
console.log( path.resolve(...p) );           // C:\pl\file.txt
console.log( path.win32.resolve(...p) );     // C:\pl\file.txt
console.log( path.posix.resolve(...p) );     // /pl/file.txt
p = ['C:\\', 'book', 'pl', '..', '', 'file.txt'];
console.log( path.resolve(...p) );           // C:\book\file.txt
```

Если указан только относительный путь, то он будет разрешен в абсолютный путь с учетом текущего рабочего каталога:

```
console.log( path.resolve('./file.txt') );
// C:\book\pl\file.txt
console.log( path.resolve('../file.txt') );
// C:\book\file.txt
console.log( path.resolve('..', '..', 'file.txt') );
// C:\file.txt
```

Если параметр не указан, функция вернет путь к текущему рабочему каталогу:

```
console.log( path.resolve() ); // C:\book\pl
```


- ❑ `relative(<Путь1>, <Путь2>)` — возвращает относительный путь от `<Путь1>` к `<Путь2>` относительно текущего рабочего каталога. Если заданы одинаковые пути, возвращается пустая строка. Если в первом параметре указана пустая строка, вместо нее будет использован путь к текущему рабочему каталогу. Пример:

```
let p1 = 'C:\\book\\p1\\test';
let p2 = 'C:\\book\\p2';
console.log( path.relative(p1, p2) );      // ..\\..\\p2
console.log( path.win32.relative(p1, p2) ); // ..\\..\\p2
console.log( path.relative('', p2) );      // ..\\p2
console.log( path.relative(p1, p1) );      // Пустая строка
```

- ❑ `parse(<Путь>)` — разделяет путь на отдельные сегменты. Возвращает объект со свойствами: `root` (обозначение диска в Windows), `dir` (путь к каталогу), `base` (имя файла или каталога), `ext` (расширение файла или пустая строка в случае каталога) и `name` (имя файла без расширения или имя каталога). Пример:

```
let p = 'C:\\book\\p1\\test.txt';
console.log( path.parse(p) );
/* {
  root: 'C:\\',
  dir: 'C:\\book\\p1',
  base: 'test.txt',
  ext: '.txt',
  name: 'test'
} */
p = 'C:\\book\\p1\\test\\';
console.log( path.win32.parse(p) );
/* {
  root: 'C:\\',
  dir: 'C:\\book\\p1',
  base: 'test',
  ext: '',
  name: 'test'
} */
p = '/book/p1/test/';
console.log( path.posix.parse(p) );
// { root: '/', dir: '/book/p1', base: 'test', ext: '',
//   name: 'test' }
```

- ❑ `format(<Объект>)` — возвращает строку с путем на основе объекта формата, выдаваемого функцией `parse()`. Значение свойства `root` игнорируется, если указано свойство `dir`. Значения свойств `name` и `ext` игнорируются, если указано свойство `base`. Пример:

```
let p = 'C:\\book\\p1\\test.txt';
let obj = path.parse(p);
console.log( obj );
```

```

/* {
  root: 'C:\\',
  dir: 'C:\\book\\p1',
  base: 'test.txt',
  ext: '.txt',
  name: 'test'
} */
console.log( path.format(obj) );           // C:\book\p1\test.txt
console.log( path.win32.format(obj) );     // C:\book\p1\test.txt
console.log( path.format({
  dir: 'C:\\book\\p1',
  base: 'test.txt'
}) );                                     // C:\book\p1\test.txt
console.log( path.format({
  root: 'C:\\',
  ext: '.txt',
  name: 'test'
}) );                                     // C:\test.txt

```

- ❑ **dirname(<Путь>)** — возвращает путь к каталогу (конечные слешы игнорируются):

```

let p = 'C:\\book\\p1\\test.txt';
console.log( path.dirname(p) );           // C:\book\p1
p = 'C:\\book\\p1\\test\\';
console.log( path.win32.dirname(p) );     // C:\book\p1
p = '/book/p1/test/';
console.log( path.posix.dirname(p) );     // /book/p1

```

- ❑ **basename(<Путь>[, <Расширение>])** — возвращает имя файла или каталога (конечные слешы игнорируются). Если указан второй параметр, указанное в нем расширение удаляется из результата. Пример:

```

let p = 'C:\\book\\p1\\test.txt';
console.log( path.basename(p) );          // test.txt
console.log( path.basename(p, '.txt') );  // test
console.log( path.basename(p, 'txt') );   // test.
console.log( path.basename(p, '.doc') );  // test.txt
p = 'C:\\book\\p1\\test\\';
console.log( path.win32.basename(p) );    // test
p = '/book/p1/test/';
console.log( path.posix.basename(p) );    // test

```

- ❑ **extname(<Путь>)** — возвращает расширение файла или пустую строку в случае каталога (конечные слешы игнорируются):

```

let p = 'C:\\book\\p1\\test.txt';
console.log( path.extname(p) );           // .txt
p = 'C:\\book\\p1\\test.js.map';
console.log( path.extname(p) );           // .map
p = 'C:\\book\\p1\\test\\';

```

```
console.log( path.win32.extname(p) );    // Пустая строка
p = '/book/pl/test.txt';
console.log( path.posix.extname(p) );    // .txt
```

ВНИМАНИЕ!

Если путь, указанный в качестве параметра, не является строкой, все функции сгенерируют исключение `TypeError`.

16.3. Запись в файл с указанием пути к файлу

Для записи в файл предназначены следующие функции из модуля `fs`:

- `writeFileSync()` — записывает данные синхронно. Если файл не существует, он будет создан, если существует — по умолчанию будет перезаписан. Если записать данные не удалось, генерируется исключение. Формат функции:

```
writeFileSync(<Путь к файлу>, <Данные>[, <Опции или кодировка>])
```

В первом параметре указывается путь к файлу в виде строки, объекта `Buffer`, объекта `URL` или дескриптора. Если указан относительный путь, он разрешается относительно текущего рабочего каталога (см. *разд. 16.1*). Чтобы задать путь относительно исполняемого файла, можно воспользоваться следующим кодом:

```
const path = require('path');
let p = path.join(__dirname, 'file1.txt');
console.log(p); // C:\book\pl\file1.txt
```

Во втором параметре указываются записываемые данные в виде строки или объекта любого класса. Если указана строка, то в третьем параметре можно передать кодировку (значение по умолчанию: `'utf8'`). Если указан объект, он будет преобразован в строку вызовом метода `toString()`. Запишем в файл строку в кодировке UTF-8 и обработаем ошибки:

```
const path = require('path');
const fs = require('fs');
try {
  let p = path.join(__dirname, 'file1.txt');
  fs.writeFileSync(p, 'Строка1', 'utf8');
  console.log('Данные успешно записаны');
} catch (e) {
  console.error(e.message);
}
console.log('Конец программы');
```

Результат в консоли (обратите внимание: инструкции выполняются последовательно, т. е. синхронно):

```
Данные успешно записаны
Конец программы
```

Пример записи объекта приведен в листинге 16.1.

Листинг 16.1. Запись объекта в файл функцией `writeFileSync()`

```
const path = require('path');
const fs = require('fs');
const obj = {
  a: 10,
  b: 20,
  toString() {
    return 'a: ' + this.a + ', b: ' + this.b;
  }
};
try {
  let p = path.join(__dirname, 'file2.txt');
  fs.writeFileSync(p, obj);
} catch (e) {
  console.error(e.message);
}
```

Содержимое файла `file2.txt`:

`a: 10, b: 20`

Чтобы записать данные в кодировке Windows-1251, следует воспользоваться пакетом `iconv-lite` (см. *разд. 15.14*):

```
const iconv = require('iconv-lite');
const fs = require('fs');
let data = iconv.encode('Строка1', 'win1251');
fs.writeFileSync('cp1251.txt', data);
console.log('Данные успешно записаны');
```

В третьем параметре можно указать объект с тремя свойствами:

- `encoding` — кодировка в виде строки или значение `null` (значение по умолчанию: `'utf8'`);
- `mode` — режим доступа к файлу в виде восьмеричного числа (значение по умолчанию: `0o666`);
- `flag` — режим открытия файла в виде строки (значение по умолчанию: `'w'`; если файл не существует, он будет создан, а если существует, он будет перезаписан).

По умолчанию файл перезаписывается. Чтобы дописать данные в файл, нужно воспользоваться функцией `appendFileSync()` или в свойстве `flag` указать значение `'a'`. Пример:

```
const fs = require('fs');
fs.writeFileSync('file1.txt', '\nСтрока2', {
  encoding: 'utf8',
  flag: 'a'
});
console.log('Данные успешно записаны');
```

Содержимое файла file1.txt:

Строка1

Строка2

- `appendFileSync()` — дописывает данные в файл синхронно. Если файл не существует, он будет создан. Если записать данные не удалось, генерируется исключение. Формат функции:

```
appendFileSync(<Путь к файлу>, <Данные>[, <Опции или кодировка>])
```

В первом параметре указывается путь к файлу в виде строки, объекта `Buffer`, объекта `URL` или дескриптора. Если указан относительный путь, он разрешается относительно текущего рабочего каталога (см. *разд. 16.1*).

Во втором параметре указываются записываемые данные в виде строки или объекта `Buffer`. Если указана строка, в третьем параметре можно передать кодировку (значение по умолчанию: `'utf8'`). Запишем строку в файл в кодировке UTF-8 и обработаем ошибки:

```
const path = require('path');
const fs = require('fs');
try {
  let p = path.join(__dirname, 'file3.txt');
  fs.appendFileSync(p, 'Строка1', 'utf8');
  console.log('Данные успешно записаны');
} catch (e) {
  console.error(e.message);
}
console.log('Конец программы');
```

Результат в консоли (обратите внимание: инструкции выполняются последовательно, т. е. синхронно):

```
Данные успешно записаны
Конец программы
```

В третьем параметре можно указать объект с тремя свойствами:

- `encoding` — кодировка в виде строки или значение `null` (значение по умолчанию: `'utf8'`);
- `mode` — режим доступа к файлу в виде восьмеричного числа (значение по умолчанию: `0o666`);
- `flag` — режим открытия файла в виде строки (значение по умолчанию: `'a'`; если файл не существует, он будет создан, а если существует, данные будут добавлены в конец содержимого файла).

Добавим строку в конец файла file3.txt:

```
const fs = require('fs');
fs.appendFileSync('file3.txt', '\nСтрока2', {encoding: 'utf8'});
console.log('Данные успешно записаны');
```

Содержимое файла file3.txt:

Строка1
Строка2

Чтобы записать данные в кодировке Windows-1251, следует воспользоваться пакетом `iconv-lite` (см. *разд. 15.14*). Добавим строку в конец файла `cp1251.txt`:

```
const iconv = require('iconv-lite');
const fs = require('fs');
let data = iconv.encode('\nСтрока2', 'win1251');
fs.appendFileSync('cp1251.txt', data);
console.log('Данные успешно записаны');
```

Содержимое файла `cp1251.txt`:

Строка1
Строка2

- ❑ `writeFile()` — записывает данные асинхронно. Если файл не существует, он будет создан, если существует — по умолчанию будет перезаписан. Формат функции:

```
writeFile(<Путь к файлу>, <Данные>[, <Опции или кодировка>], <Функция>)
```

В первом параметре указывается путь к файлу в виде строки, объекта `Buffer`, объекта `URL` или дескриптора. Если указан относительный путь, он разрешается относительно текущего рабочего каталога (см. *разд. 16.1*).

Во втором параметре указываются записываемые данные в виде строки или объекта любого класса. Если указана строка, в третьем параметре можно передать кодировку (значение по умолчанию: `'utf8'`). Если указан объект, он будет преобразован в строку вызовом метода `toString()`.

В последнем параметре указывается функция, которая будет вызвана по окончании записи вне зависимости от успешности операции. Функция принимает один параметр, через который доступен объект ошибки, если данные не удалось записать в файл, или значение `null` — если запись выполнена успешно.

Запишем строку в файл в кодировке UTF-8 и обработаем ошибки (листинг 16.2).

Листинг 16.2. Запись строки в файл функцией `writeFile()`

```
const path = require('path');
const fs = require('fs');
let p = path.join(__dirname, 'file4.txt');
fs.writeFile(p, 'Строка1', 'utf8', function(err) {
  if (err) {
    console.error(err.message);
  }
})
```

```
    else {  
        console.log('Данные успешно записаны');  
    }  
});  
console.log('Конец программы');
```

Результат в консоли:

```
Конец программы  
Данные успешно записаны
```

Обратите внимание: инструкции выполняются асинхронно. Сначала мы получили сообщение "Конец программы", а лишь затем — результат выполнения операции. Обработать результат можно только внутри функции обратного вызова.

Пример записи объекта приведен в листинге 16.4.

Листинг 16.3. Запись объекта в файл функцией `writeFile()`

```
const path = require('path');  
const fs = require('fs');  
const obj = {  
    a: 10,  
    b: 20,  
    toString() {  
        return 'a: ' + this.a + ', b: ' + this.b;  
    }  
};  
let p = path.join(__dirname, 'file5.txt');  
fs.writeFile(p, obj, 'utf8', function(err) {  
    if (err) {  
        console.error(err.message);  
    }  
});
```

Чтобы записать данные в кодировке Windows-866, следует воспользоваться пакетом `iconv-lite` (см. *разд. 15.14*):

```
const iconv = require('iconv-lite');  
const fs = require('fs');  
let data = iconv.encode('Строка1', 'cp866');  
fs.writeFile('cp866.txt', data, function(err) {  
    if (err) throw err;  
    console.log('Данные успешно записаны');  
});
```

В третьем параметре можно указать объект с тремя свойствами:

- `encoding` — кодировка в виде строки или значение `null` (значение по умолчанию: `'utf8'`);

- `mode` — режим доступа к файлу в виде восьмеричного числа (значение по умолчанию: `0o666`);
- `flag` — режим открытия файла в виде строки (значение по умолчанию: `'w'`; если файл не существует, он будет создан, а если существует, он будет перезаписан).

По умолчанию файл перезаписывается. Чтобы дописать в файл, нужно воспользоваться функцией `appendFile()` или в свойстве `flag` указать значение `'a'`. Пример:

```
const fs = require('fs');
fs.writeFile('file4.txt', '\nСтрока2', {
  encoding: 'utf8',
  flag: 'a'
}, function(err) {
  if (err) throw err;
  console.log('Данные успешно записаны');
});
```

Содержимое файла `file4.txt`:

```
Строка1
Строка2
```

- `appendFile()` — дописывает данные в файл асинхронно. Если файл не существует, он будет создан. Формат функции:

```
appendFile(<Путь к файлу>, <Данные>[, <Опции или кодировка>], <Функция>)
```

В первом параметре указывается путь к файлу в виде строки, объекта `Buffer`, объекта `URL` или дескриптора. Если указан относительный путь, он разрешается относительно текущего рабочего каталога (см. *разд. 16.1*).

Во втором параметре указываются записываемые данные в виде строки или объекта `Buffer`. Если указана строка, в третьем параметре можно передать кодировку (значение по умолчанию: `'utf8'`).

В последнем параметре указывается функция, которая будет вызвана по окончании записи вне зависимости от успешности операции. Функция принимает один параметр, через который доступен объект ошибки, если данные не удалось записать в файл, или значение `null` — если запись выполнена успешно.

Запишем строку в файл в кодировке UTF-8 и обработаем ошибки (листинг 16.4).

Листинг 16.4. Запись строки в файл функцией `appendFile()`

```
const path = require('path');
const fs = require('fs');
let p = path.join(__dirname, 'file6.txt');
fs.appendFile(p, 'Строка1', 'utf8', function(err) {
  if (err) {
    console.error(err.message);
  }
});
```



```
    else {  
        console.log('Данные успешно записаны');  
    }  
});  
console.log('Конец программы');
```

Результат в консоли:

```
Конец программы  
Данные успешно записаны
```

Обратите внимание: инструкции выполняются асинхронно. Сначала мы получили сообщение "Конец программы", а лишь затем — результат выполнения операции. Обработать результат можно только внутри функции обратного вызова.

В третьем параметре можно указать объект с тремя свойствами:

- `encoding` — кодировка в виде строки или значение `null` (значение по умолчанию: `'utf8'`);
- `mode` — режим доступа к файлу в виде восьмеричного числа (значение по умолчанию: `0o666`);
- `flag` — режим открытия файла в виде строки (значение по умолчанию: `'a'`; если файл не существует, он будет создан, а если существует, данные будут добавлены в конец файла).

Добавим строку в конец файла `file6.txt`:

```
const fs = require('fs');  
fs.appendFile('file6.txt', '\nСтрока2', {encoding: 'utf8'},  
    function(err) {  
        if (err) throw err;  
        console.log('Данные успешно записаны');  
    });
```

Содержимое файла `file6.txt`:

```
Строка1  
Строка2
```

Чтобы записать данные в кодировке Windows-866, следует воспользоваться пакетом `iconv-lite` (см. *разд. 15.14*). Добавим строку в конец файла `cp866.txt`:

```
const iconv = require('iconv-lite');  
const fs = require('fs');  
let data = iconv.encode('\nСтрока2', 'cp866');  
fs.appendFile('cp866.txt', data, function(err) {  
    if (err) throw err;  
    console.log('Данные успешно записаны');  
});
```

Содержимое файла `sr866.txt`:

Строка1

Строка2

- `promises.writeFile()` — записывает данные асинхронно. Если файл не существует, он будет создан, если существует — по умолчанию будет перезаписан. Функция возвращает объект класса `Promise` (см. *разд. 14.1*). Формат функции:

```
<Promise> = promises.writeFile(<Путь к файлу>, <Данные>[,  
                                <Опции или кодировка>])
```

В первом параметре указывается путь к файлу в виде строки, объекта `Buffer`, объекта `URL` или дескриптора (`FileHandle`). Если указан относительный путь, он разрешается относительно текущего рабочего каталога (см. *разд. 16.1*).

Во втором параметре указываются записываемые данные в виде строки, объекта класса `Buffer` или любого типизированного массива. Если указана строка, в третьем параметре можно передать кодировку (значение по умолчанию: `'utf8'`).

Запишем строку в файл в кодировке UTF-8 и обработаем ошибки (листинг 16.5).

Листинг 16.5. Запись строки в файл методом `writeFile()`

```
const path = require('path');  
const fs = require('fs');  
let p = path.join(__dirname, 'file7.txt');  
fs.promises.writeFile(p, 'Строка1', 'utf8').then(  
  function() {  
    console.log('Данные успешно записаны');  
  },  
  function(err) {  
    console.error(err.message);  
  });  
console.log('Конец программы');
```

Результат в консоли:

Конец программы

Данные успешно записаны

Обратите внимание: инструкции выполняются асинхронно. Сначала мы получили сообщение "Конец программы", а лишь затем — результат выполнения операции. Обработать результат можно только с помощью объекта `Promise`.

Чтобы записать данные в кодировке `koï8-r`, следует воспользоваться пакетом `iconv-lite` (см. *разд. 15.14*). Обработаем результат с помощью `async/await`:

```
const iconv = require('iconv-lite');  
const fs = require('fs');
```


В первом параметре указывается путь к файлу в виде строки, объекта `Buffer`, объекта `URL` или дескриптора (`FileHandle`). Если указан относительный путь, он разрешается относительно текущего рабочего каталога (см. *разд. 16.1*).

Во втором параметре указываются записываемые данные в виде строки или объекта `Buffer`. Если указана строка, в третьем параметре можно передать кодировку (значение по умолчанию: `'utf8'`).

Запишем строку в файл в кодировке UTF-8 и обработаем ошибки (листинг 16.7).

Листинг 16.7. Запись строки в файл методом `appendFile()`

```
const path = require('path');
const fs = require('fs');
let p = path.join(__dirname, 'file8.txt');
fs.promises.appendFile(p, 'Строка1', 'utf8').then(
  function() {
    console.log('Данные успешно записаны');
  },
  function(err) {
    console.error(err.message);
  });
console.log('Конец программы');
```

Результат в консоли:

```
Конец программы
Данные успешно записаны
```

Обратите внимание: инструкции выполняются асинхронно. Сначала мы получили сообщение "Конец программы", а лишь затем — результат выполнения операции. Обработать результат можно только с помощью объекта `Promise`.

В третьем параметре можно указать объект с тремя свойствами:

- `encoding` — кодировка в виде строки или значение `null` (значение по умолчанию: `'utf8'`);
- `mode` — режим доступа к файлу в виде восьмеричного числа (значение по умолчанию: `0o666`);
- `flag` — режим открытия файла в виде строки (значение по умолчанию: `'a'`; если файл не существует, он будет создан, а если существует, данные будут добавлены в конец файла).

Добавим строку в конец файла `file8.txt`:

```
const promises = require('fs').promises;
const myPromise = promises.appendFile('file8.txt', '\nСтрока2',
  {encoding: 'utf8'});
```

```
myPromise.then( function() {  
    console.log('Данные успешно записаны');  
}).catch( function(err) {  
    console.error(err.message);  
});
```

Содержимое файла file8.txt:

Строка1
Строка2

Чтобы записать данные в кодировке `koi8-r`, следует воспользоваться пакетом `iconv-lite` (см. *разд. 15.14*). Добавим строку в конец файла `koi8r.txt` и обработаем результат с помощью `async/await`:

```
const iconv = require('iconv-lite');  
const fs = require('fs');  
(async function() {  
    try {  
        let data = iconv.encode('\nСтрока2', 'koi8-r');  
        await fs.promises.appendFile('koi8r.txt', data);  
        console.log('Данные успешно записаны');  
    } catch (err) {  
        console.error(err.message);  
    }  
})();
```

Содержимое файла `koi8r.txt`:

Строка1
Строка2

16.4. Чтение из файла с указанием пути к файлу

Для чтения данных из файла предназначены следующие функции из модуля `fs`:

- ❑ `readFileSync()` — считывает все содержимое файла синхронно и возвращает его. Если файл не существует, генерируется исключение. Формат функции:

`<Строка или Buffer> = readFileSync(<Путь к файлу>[, <Опции или кодировка>])`

В первом параметре указывается путь к файлу в виде строки, объекта `Buffer`, объекта `URL` или дескриптора. При указании дескриптора чтение начинается с текущей позиции, что может не совпадать с началом файла. Если указан относительный путь, он разрешается относительно текущего рабочего каталога (см. *разд. 16.1*).

Во втором параметре можно указать кодировку или объект с двумя свойствами:

- `encoding` — кодировка в виде строки или `null` (это значение по умолчанию);
- `flag` — режим открытия файла в виде строки (значение по умолчанию: `'r'`; если файл не существует, генерируется исключение, а если существует, он открывается на чтение).

Если была указана кодировка, содержимое файла будет возвращено в виде строки, записанной в заданной кодировке. Прочитаем все содержимое файла `file1.txt` (см. *разд. 16.3*) и обработаем ошибки (листинг 16.8).

Листинг 16.8. Чтение содержимого файла функцией `readFileSync()`

```
const path = require('path');
const fs = require('fs');
try {
  let p = path.join(__dirname, 'file1.txt');
  let str = fs.readFileSync(p, 'utf8');
  // let str = fs.readFileSync(p, {encoding: 'utf8'});
  console.log(str);
} catch (e) {
  console.error(e.message);
}
console.log('Конец программы');
```

Результат в консоли (обратите внимание: выполнение инструкций происходит последовательно, т. е. синхронно):

```
Строка1
Строка2
Конец программы
```

Если кодировка не была указана, содержимое файла возвращается в виде объекта `Buffer`. Это позволяет прочитать файл в неподдерживаемой кодировке или бинарный файл. Прочитаем содержимое файла `cp1251.txt` (см. *разд. 16.3*) и преобразуем кодировку самостоятельно с помощью пакета `iconv-lite` (см. *разд. 15.14*):

```
const iconv = require('iconv-lite');
const fs = require('fs');
let buf = fs.readFileSync('cp1251.txt');
// let buf = fs.readFileSync('cp1251.txt', {encoding: null});
console.log(buf);
let str = iconv.decode(buf, 'win1251');
console.log(str);
```

Результат в консоли (мы успешно прочитали файл в кодировке Windows-1251):

```
<Buffer d1 f2 f0 ee ea e0 31 0a d1 f2 f0 ee ea e0 32>
Строка1
Строка2
```

- `readFile()` — считывает все содержимое файла асинхронно. Если файл не существует, генерируется исключение. Формат функции:

```
readFile(<Путь к файлу>[, <Опции или кодировка>], <Функция>)
```

В первом параметре указывается путь к файлу в виде строки, объекта `Buffer`, объекта `URL` или дескриптора. При указании дескриптора чтение начинается с текущей позиции, что может не совпадать с началом файла. Если указан относительный путь, он разрешается относительно текущего рабочего каталога (см. *разд. 16.1*).

Во втором параметре можно указать кодировку или объект с двумя свойствами:

- `encoding` — кодировка в виде строки или значение `null` (это значение по умолчанию);
- `flag` — режим открытия файла в виде строки (значение по умолчанию: `'r'`; если файл не существует, генерируется исключение, а если существует, он открывается на чтение).

В последнем параметре указывается функция, которая будет вызвана по окончании чтения вне зависимости от успешности операции. Функция принимает два параметра. Через первый параметр доступен объект ошибки, если не удалось прочитать файл, или значение `null` — если чтение выполнено успешно. Если ошибки не было, то через второй параметр будет доступно содержимое файла.

Если была указана кодировка, то содержимое файла будет доступно в виде строки, записанной в заданной кодировке. Прочитаем все содержимое файла `file1.txt` (см. *разд. 16.3*) и обработаем ошибки (листинг 16.9).

Листинг 16.9. Чтение содержимого файла функцией `readFile()`

```
const path = require('path');
const fs = require('fs');
let p = path.join(__dirname, 'file1.txt');
fs.readFile(p, 'utf8', function(err, str) {
// fs.readFile(p, {encoding: 'utf8'}, function(err, str) {
  if (err) {
    console.error(err.message);
  }
  else {
    console.log(str);
  }
});
console.log('Конец программы');
```

Результат в консоли:

```
Конец программы
Строка1
Строка2
```

Обратите внимание: инструкции выполняются асинхронно. Сначала мы получили сообщение "Конец программы", а лишь затем — результат выполнения операции. Обработать результат можно только внутри функции обратного вызова.

Если кодировка не была указана, содержимое файла будет представлено объектом `Buffer`. Это позволяет прочитать файл в неподдерживаемой кодировке или бинарный файл. Прочитаем содержимое файла `cp866.txt` (см. *разд. 16.3*) и преобразуем кодировку самостоятельно с помощью пакета `iconv-lite` (см. *разд. 15.14*):

```
const iconv = require('iconv-lite');
const fs = require('fs');
fs.readFile('cp866.txt', function(err, buf) {
  //fs.readFile('cp866.txt', {encoding: null}, function(err, buf) {
    if (err) throw err;
    console.log(buf);
    let str = iconv.decode(buf, 'cp866');
    console.log(str);
  });
```

Результат в консоли (мы успешно прочитали файл в кодировке Windows-866):

```
<Buffer 91 e2 e0 ae aa a0 31 0a 91 e2 e0 ae aa a0 32>
Строка1
Строка2
```

- `promises.readFile()` — считывает все содержимое файла асинхронно. Если файл не существует, генерируется исключение. Функция возвращает объект `Promise` (см. *разд. 14.1*). Формат функции:

```
<Promise> = promises.readFile(<Путь к файлу>[, <Опции или кодировка>])
```

В первом параметре указывается путь к файлу в виде строки, объекта `Buffer`, объекта `URL` или дескриптора (`FileHandle`). При указании дескриптора чтение начинается с текущей позиции, что может не совпадать с началом файла. Если указан относительный путь, он разрешается относительно текущего рабочего каталога (см. *разд. 16.1*).

Во втором параметре можно указать кодировку или объект с двумя свойствами:

- `encoding` — кодировка в виде строки или значение `null` (это значение по умолчанию);
- `flag` — режим открытия файла в виде строки (значение по умолчанию: `'r'`; если файл не существует, генерируется исключение, а если существует, он открывается на чтение).

Если была указана кодировка, содержимое файла будет доступно в виде строки, записанной в заданной кодировке. Прочитаем все содержимое файла `file1.txt` (см. *разд. 16.3*) и обработаем ошибки (листинг 16.10).

Листинг 16.10. Чтение содержимого файла методом `readFile()`

```
const path = require('path');
const fs = require('fs');
let p = path.join(__dirname, 'file1.txt');
```



```

fs.promises.readFile(p, 'utf8').then(
// fs.promises.readFile(p, {encoding: 'utf8'}).then(
  function(str) {
    console.log(str);
  },
  function(err) {
    console.error(err.message);
  });
console.log('Конец программы');

```

Результат в консоли:

```

Конец программы
Строка1
Строка2

```

Обратите внимание: инструкции выполняются асинхронно. Сначала мы получили сообщение "Конец программы", а лишь затем — результат выполнения операции. Обработать результат можно только с помощью объекта `Promise`.

Если кодировка не была указана, содержимое файла будет представлено объектом `Buffer`. Это позволяет прочитать файл в не поддерживаемой Node.js кодировке или бинарный файл. Прочитаем содержимое файла `koi8r.txt` (см. *разд. 16.3*) и преобразуем кодировку самостоятельно с помощью пакета `iconv-lite` (см. *разд. 15.14*):

```

const iconv = require('iconv-lite');
const fs = require('fs');
(async function() {
  try {
    let buf = await fs.promises.readFile('koi8r.txt');
    // let buf = await fs.promises.readFile('koi8r.txt',
    //                                     {encoding: null});
    console.log(buf);
    let str = iconv.decode(buf, 'koi8-r');
    console.log(str);
  } catch (err) {
    console.error(err.message);
  }
})();

```

Результат в консоли (мы успешно прочитали файл в кодировке koi8-r):

```

<Buffer f3 d4 d2 cf cb c1 31 0a f3 d4 d2 cf cb c1 32>
Строка1
Строка2

```

НА ЗАМЕТКУ

Если в параметре <Путь к файлу> указан дескриптор, файл не закрывается автоматически, а чтение выполняется с текущей позиции.

16.5. Открытие и закрытие файла

Для открытия файла служат следующие функции из модуля `fs`:

- `openSync()` — открывает файл синхронно и возвращает его дескриптор в виде целого числа. Если файл открыть не удалось, генерируется исключение. Формат функции:

```
<Дескриптор> = openSync(<Путь к файлу>[, <Режим открытия>[,  
                        <Режим доступа>]])
```

В первом параметре задается путь к файлу в виде строки, объекта `Buffer` или `URL`. Если указан относительный путь, он разрешается относительно текущего рабочего каталога (см. *разд. 16.1*). Если нужно указать путь относительно исполняемого файла, можно воспользоваться следующим кодом:

```
const path = require('path');  
let p = path.join(__dirname, 'file1.txt');  
console.log(p); // C:\book\p1\file1.txt
```

Во втором параметре задается режим открытия файла в виде строки или числа. Значение по умолчанию: `'r'` (если файл не существует, генерируется исключение, а если существует, он открывается на чтение).

В третьем параметре можно указать режим доступа к файлу (если файл создается) в виде восьмеричного числа (значение по умолчанию: `0o666`).

После окончания работы с файлом его необходимо закрыть с помощью функции `closeSync(<Дескриптор>)`. Обычно закрытие файла выполняется внутри блока `finally` — в этом случае файл будет закрыт даже при возникновении исключения внутри блока `try`.

Дескриптор, возвращаемый функцией `openSync()`, можно передать функциям `writeFileSync()`, `appendFileSync()` и `readFileSync()`, рассмотренным в *разд. 16.3* и *16.4*. Но при этом режим открытия файла должен совпадать с предназначением функции, т. к. нельзя записать строку в файл, открытый только для чтения. Откроем файл `file1.txt` (см. *разд. 16.3*) на чтение, прочитаем его содержимое вызовом функции `readFileSync()`, закроем файл и обработаем ошибки (листинг 16.11).

Листинг 16.11. Синхронные открытие, чтение и закрытие файла

```
const fs = require('fs');  
let fd;  
try {  
    fd = fs.openSync('file1.txt', 'r');  
    console.log(`Файл успешно открыт. fd = ${fd}`);  
    let str = fs.readFileSync(fd, 'utf8');  
    console.log(str);  
} catch (err) {  
    console.error(err.message);  
}
```

```
} finally {  
    if (fd !== undefined) {  
        fs.closeSync(fd);  
        console.log('Файл закрыт');  
    }  
}  
  
console.log('Конец программы');
```

Результат в консоли (обратите внимание: инструкции выполняются последовательно, т. е. синхронно):

```
Файл успешно открыт. fd = 3  
Строка1  
Строка2  
Файл закрыт  
Конец программы
```

□ `open()` — открывает файл асинхронно. Формат функции:

```
open(<Путь к файлу>[, <Режим открытия>[, <Режим доступа>]], <Функция>)
```

В первом параметре указывается путь к файлу в виде строки, объекта `Buffer` или `URL`. Если указан относительный путь, он разрешается относительно текущего рабочего каталога (см. *разд. 16.1*).

Во втором параметре задается режим открытия файла в виде строки или числа. Значение по умолчанию: `'r'` (если файл не существует, генерируется исключение, а если существует, он открывается на чтение).

В третьем параметре можно указать режим доступа к файлу (если файл создается) в виде восьмеричного числа (значение по умолчанию: `0o666`).

В последнем параметре указывается функция, которая будет вызвана вне зависимости от успешности операции. Функция принимает два параметра. Через первый параметр доступен объект ошибки, если файл открыть не удалось, или значение `null` — если файл открыт. Через второй параметр доступен дескриптор файла в виде целого числа.

После окончания работы с файлом его необходимо закрыть с помощью функции `close(<Дескриптор>, <Функция>)`. В первом параметре указывается дескриптор открытого ранее файла, а во втором параметре — функция обратного вызова, внутри которой через параметр будет доступен объект ошибки или значение `null`, если файл был успешно закрыт.

Дескриптор, возвращаемый функцией `open()`, можно передать функциям `writeFile()`, `appendFile()` и `readFile()`, рассмотренным в *разд. 16.3* и *16.4*. Но при этом режим открытия файла должен совпадать с предназначением функции, т. к. нельзя записать строку в файл, открытый только для чтения. Откроем файл `sr1251.txt` (см. *разд. 16.3*) на чтение, прочитаем его содержимое с помощью функции `readFile()`, закроем файл и обработаем ошибки (листинг 16.12).

Листинг 16.12. Асинхронное открытие, чтение и закрытие файла 1

```
const iconv = require('iconv-lite');
const fs = require('fs');
function closeFd(fd) {
  if (typeof fd === 'number') {
    fs.close(fd, function(err) {
      if (err) console.error(err.message);
      else console.log('Файл закрыт');
    });
  }
}
fs.open('cpl251.txt', 'r', function(err, fd) {
  if (err) {
    console.error(err.message);
    return;
  }
  console.log(`Файл успешно открыт. fd = ${fd}`);
  fs.readFile(fd, function(err, buf) {
    if (err) {
      console.error(err.message);
      closeFd(fd);
      return;
    }
    try {
      let str = iconv.decode(buf, 'win1251');
      console.log(str);
    } finally {
      closeFd(fd);
    }
  });
});
console.log('Конец программы');
```

Результат в консоли:

```
Конец программы
Файл успешно открыт. fd = 3
Строка1
Строка2
Файл закрыт
```

Обратите внимание: инструкции выполняются асинхронно. Сначала мы получили сообщение "Конец программы", а лишь затем — результат выполнения операции. Обработать результат можно только внутри функции обратного вызова;

- `promises.open()` — открывает файл асинхронно. Функция возвращает объект `Promise` (см. *разд. 14.1*), который при успешном открытии файла содержит объект `FileHandle`. Формат функции:

```
<Promise> = promises.open(<Путь к файлу>[, <Режим открытия>[,  
                           <Режим доступа>]])
```

В первом параметре указывается путь к файлу в виде строки, объекта `Buffer` или URL. Если указан относительный путь, он разрешается относительно текущего рабочего каталога (см. *разд. 16.1*).

Во втором параметре задается режим открытия файла в виде строки или числа. Значение по умолчанию: `'r'` (если файл не существует, генерируется исключение, а если существует, он открывается на чтение).

В третьем параметре можно указать режим доступа к файлу (если файл создается) в виде восьмеричного числа (значение по умолчанию: `0o666`).

После окончания работы с файлом его необходимо обязательно закрыть с помощью метода `<FileHandle>.close()`.

Объект `FileHandle` можно передать функциям `promises.writeFile()`, `promises.appendFile()` и `promises.readFile()`, рассмотренным в *разд. 16.3* и *16.4*. Но при этом режим открытия файла должен совпадать с предназначением функции, т. к. нельзя записать строку в файл, открытый только для чтения. Откроем файл `cp866.txt` (см. *разд. 16.3*) на чтение, прочитаем его содержимое с помощью функции `promises.readFile()`, закроем файл и обработаем ошибки (листинг 16.13).

Листинг 16.13. Асинхронные открытие, чтение и закрытие файла 2

```
const iconv = require('iconv-lite');
const fs = require('fs');
(async function() {
  let fh;
  try {
    fh = await fs.promises.open('cp866.txt', 'r');
    console.log('Файл успешно открыт');
    let buf = await fs.promises.readFile(fh);
    let str = iconv.decode(buf, 'cp866');
    console.log(str);
  } catch (err) {
    console.error(err.message);
  } finally {
    if (fh !== undefined) {
      await fh.close();
      console.log('Файл закрыт');
    }
  }
})();
console.log('Конец программы');
```

Результат в консоли:

Конец программы

Файл успешно открыт

```
Строка1  
Строка2  
Файл закрыт
```

Обратите внимание: инструкции выполняются асинхронно. Сначала мы получили сообщение "Конец программы", а лишь затем результат выполнения операции. Обработать результат можно только с помощью объекта `Promise`.

16.6. Режимы открытия файла

Параметр <Режим открытия> в функциях открытия файла может принимать следующие строковые значения:

- ☐ `r` — только чтение. Курсор устанавливается на начало файла. Если файл не существует, генерируется исключение;
- ☐ `r+` — чтение и запись. Курсор устанавливается на начало файла. Если файл не существует, генерируется исключение;
- ☐ `rs+` — чтение и запись в синхронном режиме. Указывает операционной системе не использовать кэш. Курсор устанавливается на начало файла. Если файл не существует, генерируется исключение;
- ☐ `w` — запись. Если файл не существует, он будет создан. Если файл существует, он будет перезаписан. Курсор устанавливается на начало файла;
- ☐ `wx` — запись. Если файл не существует, он будет создан. Если файл существует, генерируется исключение. Курсор устанавливается на начало файла;
- ☐ `w+` — чтение и запись. Если файл не существует, он будет создан. Если файл существует, он будет перезаписан. Курсор устанавливается на начало файла;
- ☐ `wx+` — чтение и запись. Если файл не существует, он будет создан. Если файл существует, генерируется исключение. Курсор устанавливается на начало файла;
- ☐ `a` — дозапись в конец файла. Если файл не существует, он будет создан;
- ☐ `ax` — запись. Если файл не существует, он будет создан. Если файл существует, генерируется исключение;
- ☐ `as` — дозапись в конец файла в синхронном режиме. Если файл не существует, он будет создан;
- ☐ `a+` — чтение с начала файла и дозапись в его конец. Если файл не существует, он будет создан;
- ☐ `ax+` — чтение и запись. Если файл не существует, он будет создан. Если файл существует, генерируется исключение;
- ☐ `as+` — чтение с начала файла и дозапись в его конец в синхронном режиме. Если файл не существует, он будет создан.

Еще в параметре <Режим открытия> могут быть указаны следующие основные флаги или их комбинация через оператор `|` (константы из `fs.constants`):

- ☐ `O_RDONLY` — только чтение;
- ☐ `O_WRONLY` — только запись;
- ☐ `O_RDWR` — чтение и запись;
- ☐ `O_APPEND` — дозапись в конец файла;
- ☐ `O_CREAT` — создать файл, если он не существует;
- ☐ `O_CREAT | O_EXCL` — создать файл. Если файл существует, генерируется исключение;
- ☐ `O_TRUNC` — очистить содержимое файла.

ПРИМЕЧАНИЕ

Полный список флагов можно найти в документации.

Пример открытия файла на чтение приведен в листинге 16.14.

Листинг 16.14. Открытие файла на чтение

```
const fs = require('fs');
let fd;
try {
  fd = fs.openSync('file1.txt', fs.constants.O_RDONLY);
  console.log(`Файл успешно открыт. fd = ${fd}`);
  let str = fs.readFileSync(fd, 'utf8');
  console.log(str);
} catch (err) {
  console.error(err.message);
} finally {
  if (fd !== undefined) {
    fs.closeSync(fd);
    console.log('Файл закрыт');
  }
}
```

Откроем файл на запись и запишем в него строку. Если файл не существует, создадим его. Если файл существует, то очистим его (аналог режима `'w'`). Готовый код приведен в листинге 16.15.

Листинг 16.15. Открытие файла на запись

```
const fs = require('fs');
let fd;
try {
  let flags = fs.constants.O_WRONLY | fs.constants.O_CREAT |
    fs.constants.O_TRUNC;
```

```
fd = fs.openSync('file2.txt', flags);
console.log(`Файл успешно открыт. fd = ${fd}`);
fs.writeFileSync(fd, 'Строка1', 'utf8');
} catch (err) {
  console.error(err.message);
} finally {
  if (fd !== undefined) {
    fs.closeSync(fd);
    console.log('Файл закрыт');
  }
}
```

16.7. Запись в файл с указанием дескриптора

Для записи в уже открытый файл с указанным дескриптором предназначены следующие функции:

- `writeSync()` — записывает данные в файл, открытый в синхронном режиме. Возвращает число записанных байтов. Форматы функции:

```
writeSync(<Дескриптор>, <Строка>[, <Позиция>[, <Кодировка>]])
writeSync(<Дескриптор>, <Buffer>[, <Начало>[, <Длина>[, <Позиция>]]])
```

Если во втором параметре указана строка, то в параметре `<Кодировка>` можно задать кодировку (значение по умолчанию: `'utf8'`). Если во втором параметре указаны данные в виде объекта `Buffer`, `DataView` или типизированного массива, то в параметре `<Начало>` можно задать начальный индекс, а в параметре `<Длина>` — число считываемых из массива байтов. Если указан параметр `<Позиция>`, то данные записываются в файл, начиная с заданной позиции от начала файла (в некоторых случаях значение может быть проигнорировано, например при использовании режима `'a'`). Если тип этого параметра не является числом, запись выполняется в текущую позицию.

Пример записи в файл приведен в листинге 16.16.

Листинг 16.16. Запись в файл функцией `writeSync()`

```
const fs = require('fs');
let fd;
try {
  fd = fs.openSync('file2.txt', 'w');
  console.log(`Файл успешно открыт. fd = ${fd}`);
  console.log( fs.writeFileSync(fd, 'Строка1') );           // 13
  let buf = Buffer.from('\nСтрока2', 'utf8');
  console.log( fs.writeFileSync(fd, buf) );                 // 14
} catch (err) {
  console.error(err.message);
}
```



```

} finally {
  if (fd !== undefined) {
    fs.closeSync(fd);
    console.log('Файл закрыт');
  }
}
console.log('Конец программы');

```

Результат в консоли (обратите внимание: инструкции выполняются последовательно, т. е. синхронно):

```

Файл успешно открыт. fd = 3
13
14
Файл закрыт
Конец программы

```

- `writevSync()` — записывает данные в файл синхронно сразу из нескольких источников. Возвращает число записанных байтов. Формат функции:

```
writevSync(<Дескриптор>, <Массив>[, <Позиция>])
```

Во втором параметре задается массив с объектами `Buffer`, `DataView` или типизированными массивами. Если указан параметр `<Позиция>`, то данные записываются в файл, начиная с заданной позиции от начала файла (в некоторых случаях значение может быть проигнорировано, например при использовании режима `'a'`). Если тип этого параметра не является числом, запись выполняется в текущую позицию.

Пример записи в файл приведен в листинге 16.17.

Листинг 16.17. Запись в файл функцией `writevSync()`

```

const fs = require('fs');
let fd;
try {
  fd = fs.openSync('file2.txt', 'w');
  console.log(`Файл успешно открыт. fd = ${fd}`);
  let buf1 = Buffer.from('Строка1', 'utf8');
  let buf2 = Buffer.from('\nСтрока2', 'utf8');
  console.log( fs.writevSync(fd, [buf1, buf2]) ); // 27
} catch (err) {
  console.error(err.message);
} finally {
  if (fd !== undefined) {
    fs.closeSync(fd);
    console.log('Файл закрыт');
  }
}
console.log('Конец программы');

```

Результат в консоли (обратите внимание: инструкции выполняются последовательно, т. е. синхронно):

```
Файл успешно открыт. fd = 3
27
Файл закрыт
Конец программы
```

□ `write()` — записывает данные в файл асинхронно. Форматы функции:

```
write(<Дескриптор>, <Строка>[, <Позиция>[, <Кодировка>]], <Функция>)
write(<Дескриптор>, <Buffer>[, <Начало>[, <Длина>[, <Позиция>]]], <Функция>)
```

Если во втором параметре указана строка, то в параметре `<Кодировка>` можно задать кодировку (значение по умолчанию: `'utf8'`). Если во втором параметре указаны данные в виде объекта `Buffer`, `DataView` или типизированного массива, то в параметре `<Начало>` можно задать начальный индекс, а в параметре `<Длина>` — число считываемых из массива байтов. Если указан параметр `<Позиция>`, то данные записываются в файл, начиная с заданной позиции от начала файла (в некоторых случаях значение может быть проигнорировано, например при использовании режима `'a'`). Если тип этого параметра не является числом, запись выполняется в текущую позицию.

Во втором параметре также можно указать любой объект с методом `toString()`. В этом случае в файл будет записан результат, возвращенный этим методом.

В последнем параметре указывается функция, которая будет вызвана по окончании записи вне зависимости от успешности операции. Функция принимает три параметра. Через первый параметр доступен объект ошибки, если данные не удалось записать в файл, или значение `null` — если запись выполнена успешно. Через второй параметр доступно число записанных байтов, а через третий — строка (при использовании первого формата) или объекты `Buffer`, `DataView` или типизированный массив (при использовании второго формата).

Пример записи в файл приведен в листинге 16.18.

Листинг 16.18. Запись строк в файл функцией `write()`

```
const fs = require('fs');
function closeFd(fd) {
  if (typeof fd === 'number') {
    fs.close(fd, function(err) {
      if (err) console.error(err.message);
      else console.log('Файл закрыт');
    });
  }
}
fs.open('file2.txt', 'w', function(err, fd) {
  if (err) {
    console.error(err.message);
    return;
  }
}
```

```

console.log(`Файл успешно открыт. fd = ${fd}`);
fs.write(fd, 'Строка1', function(err, w, s) {
  if (err) {
    console.error(err.message);
    closeFd(fd);
    return;
  }
  console.log(`w = ${w}`);
  console.log(`s =`, s);
  let buf = Buffer.from('\nСтрока2', 'utf8');
  fs.write(fd, buf, 0, buf.length, function(e, bw, b) {
    if (e) {
      console.error(e.message);
      closeFd(fd);
      return;
    }
    console.log(`bw = ${bw}`);
    console.log(`b =`, b);
    closeFd(fd);
  });
});
});
console.log('Конец программы');

```

Результат в консоли:

```

Конец программы
Файл успешно открыт. fd = 3
w = 13
s = Строка1
bw = 14
b = <Buffer 0a d0 a1 d1 82 d1 80 d0 be d0 ba d0 b0 32>
Файл закрыт

```

Обратите внимание: инструкции выполняются асинхронно. Сначала мы получили сообщение "Конец программы", а лишь затем — результат выполнения операции. Обработать результат можно только внутри функции обратного вызова. Нельзя вызывать функцию `write()` повторно, не дождавшись выполнения функции обратного вызова. В листинге 16.18 повторный вызов происходит внутри обработчика первого вызова функции `write()`.

Пример записи объекта приведен в листинге 16.19.

Листинг 16.19. Запись объекта в файл функцией `write()`

```

const fs = require('fs');
const obj = {
  a: 10,
  b: 20,

```

```
    toString() {
      return 'a: ' + this.a + ', b: ' + this.b;
    }
  };
  fs.open('file3.txt', 'w', function(err, fd) {
    if (err) {
      return;
    }
    fs.write(fd, obj, function(err, w, s) {
      fs.close(fd);
    });
  });
});
```

Содержимое файла file3.txt:

a: 10, b: 20

- `writev()` — записывает данные в файл асинхронно сразу из нескольких источников. Формат функции:

`writev(<Дескриптор>, <Массив>[, <Позиция>], <Функция>)`

Во втором параметре задается массив с объектами `Buffer`, `DataView` или типизированными массивами. Если указан параметр `<Позиция>`, то данные записываются в файл, начиная с заданной позиции от начала файла (в некоторых случаях значение может быть проигнорировано, например при использовании режима `'a'`). Если тип параметра не является числом, запись выполняется в текущую позицию.

В последнем параметре указывается функция, которая будет вызвана по окончании записи вне зависимости от успешности операции. Функция принимает три параметра. Через первый параметр доступен объект ошибки, если данные не удалось записать в файл, или значение `null` — если запись выполнена успешно. Через второй параметр доступно число записанных байтов, а через третий — массив с объектами источников.

Пример записи в файл приведен в листинге 16.20.

Листинг 16.20. Запись в файл функцией `writev()`

```
const fs = require('fs');
function closeFd(fd) {
  if (typeof fd === 'number') {
    fs.close(fd, function(err) {
      if (err) console.error(err.message);
      else console.log('Файл закрыт');
    });
  }
}
```

```

fs.open('file2.txt', 'w', function(err, fd) {
  if (err) {
    console.error(err.message);
    return;
  }
  console.log(`Файл успешно открыт. fd = ${fd}`);
  let buf1 = Buffer.from('Строка1', 'utf8');
  let buf2 = Buffer.from('\nСтрока2', 'utf8');
  fs.writev(fd, [buf1, buf2], function(err, bw, bs) {
    if (err) {
      console.error(err.message);
      closeFd(fd);
      return;
    }
    console.log(`bw = ${bw}`);
    console.log('bs =', bs);
    closeFd(fd);
  });
});
console.log('Конец программы');

```

Результат в консоли:

```

Конец программы
Файл успешно открыт. fd = 3
bw = 27
bs = [
  <Buffer d0 a1 d1 82 d1 80 d0 be d0 ba d0 b0 31>,
  <Buffer 0a d0 a1 d1 82 d1 80 d0 be d0 ba d0 b0 32>
]
Файл закрыт

```

Обратите внимание: инструкции выполняются асинхронно. Сначала мы получили сообщение "Конец программы", а лишь затем — результат выполнения операции. Обработать результат можно только внутри функции обратного вызова;

- `<FileHandle>.write()` — записывает данные в файл асинхронно. Форматы метода:

```

<Promise> = <FileHandle>.write(<Строка>[, <Позиция>[, <Кодировка>]])
<Promise> = <FileHandle>.write(<Buffer>[, <Начало>[, <Длина>[,
  <Позиция>]]])

```

В качестве `<FileHandle>` указывается дескриптор файла, открытого с помощью функции `promises.open()`. Если в первом параметре указана строка, то в параметре `<Кодировка>` можно задать кодировку (значение по умолчанию: `'utf8'`). Если в первом параметре данные указаны в виде объекта `Buffer` или `Uint8Array`, то в параметре `<Начало>` можно задать начальный индекс, а в параметре `<Длина>` — число считываемых из массива байтов. Если указан параметр `<Позиция>`,

то данные записываются в файл, начиная с заданной позиции от начала файла (в некоторых случаях значение может быть проигнорировано, например при использовании режима 'a'). Если тип параметра не является числом, то запись выполняется в текущую позицию.

Метод возвращает объект `Promise`, содержащий объект с двумя свойствами. Через свойство `bytesWritten` доступно число записанных байтов, а через свойство `buffer` — строка (при использовании первого формата) или объект `Buffer` или `Uint8Array` (при использовании второго формата).

Пример записи в файл приведен в листинге 16.21.

Листинг 16.21. Запись в файл методом `write()`

```
const fs = require('fs');
(async function() {
  let fh;
  try {
    fh = await fs.promises.open('file2.txt', 'w');
    console.log('Файл успешно открыт');
    let result = await fh.write('Строка1');
    console.log(result);
    let buf = Buffer.from('\nСтрока2', 'utf8');
    result = await fh.write(buf);
    console.log(result);
  } catch (err) {
    console.error(err.message);
  } finally {
    if (fh !== undefined) {
      await fh.close();
      console.log('Файл закрыт');
    }
  }
})();
console.log('Конец программы');
```

Результат в консоли:

```
Конец программы
Файл успешно открыт
{ bytesWritten: 13, buffer: 'Строка1' }
{
  bytesWritten: 14,
  buffer: <Buffer 0a d0 a1 d1 82 d1 80 d0 be d0 ba d0 b0 32>
}
Файл закрыт
```

Обратите внимание: инструкции выполняются асинхронно. Сначала мы получили сообщение "Конец программы", а лишь затем — результат выполнения операции. Обработать результат можно только с помощью объекта `Promise`;

- ❑ `<FileHandle>.writev()` — записывает данные в файл асинхронно сразу из нескольких источников. Формат метода:

```
<Promise> = <FileHandle>.writev(<Массив>[, <Позиция>])
```

В качестве `<FileHandle>` указывается дескриптор файла, открытого с помощью функции `promises.open()`. В первом параметре задается массив с объектами `Buffer`, `DataView` или типизированными массивами. Если указан параметр `<Позиция>`, то данные записываются в файл, начиная с заданной позиции от начала файла (в некоторых случаях значение может быть проигнорировано, например при использовании режима `'a'`). Если тип параметра не является числом, то запись выполняется в текущую позицию.

Метод возвращает объект `Promise`, содержащий объект с двумя свойствами. Через свойство `bytesWritten` доступно число записанных байтов, а через свойство `buffers` — массив с объектами источников.

Пример записи в файл приведен в листинге 16.22.

Листинг 16.22. Запись в файл методом `writev()`

```
const fs = require('fs');
(async function() {
  let fh;
  try {
    fh = await fs.promises.open('file2.txt', 'w');
    console.log('Файл успешно открыт');
    let buf1 = Buffer.from('Строка1', 'utf8');
    let buf2 = Buffer.from('\nСтрока2', 'utf8');
    let result = await fh.writev([buf1, buf2]);
    console.log(result);
  } catch (err) {
    console.error(err.message);
  } finally {
    if (fh !== undefined) {
      await fh.close();
      console.log('Файл закрыт');
    }
  }
})();
console.log('Конец программы');
```

Результат в консоли:

```
Конец программы
Файл успешно открыт
{
  bytesWritten: 27,
```

```

    buffers: [
      <Buffer d0 a1 d1 82 d1 80 d0 be d0 ba d0 b0 31>,
      <Buffer 0a d0 a1 d1 82 d1 80 d0 be d0 ba d0 b0 32>
    ]
  }
}
Файл закрыт

```

Обратите внимание: инструкции выполняются асинхронно. Сначала мы получили сообщение "Конец программы", а лишь затем результат выполнения операции. Обработать результат можно только с помощью объекта `Promise`;

- `<FileHandle>.writeFile()` и `<FileHandle>.appendFile()` — записывают данные в файл в текущую позицию курсора асинхронно. Форматы методов:

```

<Promise> = <FileHandle>.writeFile(<Строка или Buffer>[,
                                   <Опции или кодировка>])
<Promise> = <FileHandle>.appendFile(<Строка или Buffer>[,
                                     <Опции или кодировка>])

```

В качестве `<FileHandle>` указывается дескриптор файла, открытого с помощью функции `promises.open()`. Если в первом параметре указана строка, то в параметре `<Кодировка>` можно задать кодировку в виде строки (значение по умолчанию: `'utf8'`) или в качестве значения свойства `encoding` объекта (`{encoding: 'utf8'}`). Если в первом параметре указаны данные в виде объекта `Buffer` или `Uint8Array`, то кодировка игнорируется. Методы возвращают объект `Promise`.

Пример записи в файл приведен в листинге 16.23.

Листинг 16.23. Запись в файл методом `writeFile()`

```

const fs = require('fs');
(async function() {
  let fh;
  try {
    fh = await fs.promises.open('file2.txt', 'w');
    console.log('Файл успешно открыт');
    await fh.writeFile('Тест', 'utf8');
    console.log('Данные записаны');
  } catch (err) {
    console.error(err.message);
  } finally {
    if (fh !== undefined) {
      await fh.close();
      console.log('Файл закрыт');
    }
  }
})();
console.log('Конец программы');

```


Результат в консоли:

```
Конец программы  
Файл успешно открыт  
Данные записаны  
Файл закрыт
```

Обратите внимание: инструкции выполняются асинхронно. Сначала мы получили сообщение "Конец программы", а лишь затем — результат выполнения операции. Обработать результат можно только с помощью объекта `Promise`.

16.8. Чтение из файла с указанием дескриптора

Для чтения из файла, открытого ранее, с указанным дескриптором предназначены следующие функции:

- `readSync()` — считывает данные из файла синхронно. Возвращает число считанных байтов. Форматы функции:

```
readSync(<Дескриптор>, <Buffer>, <Начало>, <Длина>, <Позиция>)  
readSync(<Дескриптор>, <Buffer>[, <Опции>])
```

Во втором параметре указывается буфер, в котором будут сохраняться считанные данные, — объекты `Buffer`, `DataView` или типизированный массив. В параметре `<Начало>` указывается начальный индекс внутри буфера, а в параметре `<Длина>` — число байтов, доступных для записи в буфер. В параметре `<Позиция>` задается позиция от начала файла, с которой начинается считывание. Если параметр имеет значение `null`, то считывание начинается с текущей позиции курсора и позиция курсора после завершения чтения будет обновлена. Если в параметре указано число, то позиция курсора не изменится.

В параметре `<Опции>` можно указать объект со следующими свойствами:

- `offset` — начальный индекс внутри буфера (значение по умолчанию: 0);
- `length` — число байтов, доступных для записи в буфер (значение по умолчанию: размер буфера);
- `position` — позиция от начала файла, с которой начинается считывание, или значение `null` (значение по умолчанию: `null`).

Считаем из файла `file1.txt` (см. *разд. 16.3*) первые 12 байтов (листинг 16.24).

Листинг 16.24. Чтение из файла функцией `readSync()` 1

```
const fs = require('fs');  
let fd;  
try {  
  fd = fs.openSync('file1.txt', 'r');  
  console.log(`Файл успешно открыт. fd = ${fd}`);  
}
```

```
let buf = Buffer.alloc(12);
let n = fs.readFileSync(fd, buf, 0, buf.length, null);
console.log(n);
let str = buf.toString('utf8', 0, n);
console.log(str);
} catch (err) {
  console.error(err.message);
} finally {
  if (fd !== undefined) {
    fs.closeSync(fd);
    console.log('Файл закрыт');
  }
}
console.log('Конец программы');
```

Результат в консоли:

```
Файл успешно открыт. fd = 3
12
Строка
Файл закрыт
Конец программы
```

Если нужно прочитать весь файл целиком, можно воспользоваться функцией `readFileSync()` (см. *разд. 16.5*) или сначала получить размер файла с помощью функции `statSync()`. Пример второго варианта, а заодно и пример использования второго формата функции `readSync()` приведен в листинге 16.25.

Листинг 16.25. Чтение из файла функцией `readSync()` 2

```
const fs = require('fs');
let fd;
try {
  let p = 'file1.txt';
  let stats = fs.statSync(p);
  fd = fs.openSync(p, 'r');
  console.log(`Файл успешно открыт. fd = ${fd}`);
  let buf = Buffer.alloc(stats.size);
  let n = fs.readSync(fd, buf, {
    offset: 0, length: buf.length, position: null
  });
  console.log(n);
  let str = buf.toString('utf8', 0, n);
  console.log(str);
} catch (err) {
  console.error(err.message);
}
```

```

} finally {
  if (fd !== undefined) {
    fs.closeSync(fd);
    console.log('Файл закрыт');
  }
}
console.log('Конец программы');

```

Результат в консоли:

```

Файл успешно открыт. fd = 3
27
Строка1
Строка2
Файл закрыт
Конец программы

```

- `readvSync()` — считывает данные из файла синхронно и записывает их в несколько буферов. Возвращает число прочитанных байтов. Формат функции:

```
readvSync(<Дескриптор>, <Массив>[, <Позиция>])
```

В параметре `<Массив>` указывается массив с объектами `Buffer`, `DataView` или типизированными массивами. Если указан параметр `<Позиция>`, то данные считываются, начиная с заданной позиции от начала файла. Если параметр не указан или его тип не является числом, то чтение выполняется с текущей позиции.

С помощью кода, приведенного в листинге 16.26, прочитаем содержимое файла `file1.txt` (см. *разд. 16.3*).

Листинг 16.26. Чтение из файла функцией `readvSync()`

```

const fs = require('fs');
let fd;
try {
  fd = fs.openSync('file1.txt', 'r');
  console.log(`Файл успешно открыт. fd = ${fd}`);
  let buf1 = Buffer.alloc(15);
  let buf2 = Buffer.alloc(15);
  let n = fs.readvSync(fd, [buf1, buf2]);
  console.log(n);
  console.log(buf1);
  console.log(buf2);
} catch (err) {
  console.error(err.message);
} finally {
  if (fd !== undefined) {
    fs.closeSync(fd);
  }
}

```

```
        console.log('Файл закрыт');
    }
}
console.log('Конец программы');
```

Результат в консоли:

```
Файл успешно открыт. fd = 3
27
<Buffer d0 a1 d1 82 d1 80 d0 be d0 ba d0 b0 31 0a d0>
<Buffer a1 d1 82 d1 80 d0 be d0 ba d0 b0 32 00 00 00>
Файл закрыт
Конец программы
```

❑ `read()` — считывает данные из файла асинхронно. Форматы функции:

```
read(<Дескриптор>, <Buffer>, <Начало>, <Длина>, <Позиция>, <Функция>)
read(<Дескриптор>[, <Опции>], <Функция>)
```

В параметре `<Buffer>` должен быть указан буфер для хранения считанных данных — объекты `Buffer`, `DataView` или типизированный массив. В параметре `<Начало>` указывается начальный индекс внутри буфера, а в параметре `<Длина>` — число байтов, доступных для записи в буфер. В параметре `<Позиция>` задается позиция от начала файла, с которой начинается считывание. Если параметр имеет значение `null`, то считывание начинается с текущей позиции курсора, и позиция курсора после завершения чтения будет обновлена. Если в параметре указано число, то позиция курсора не изменится.

В параметре `<Опции>` можно указать объект со следующими свойствами:

- `buffer` — буфер для сохранения считанных данных в виде объектов `Buffer`, `DataView` или типизированного массива (значение по умолчанию: объект `Buffer` размером 16 384 байт, или 16 Кбайт);
- `offset` — начальный индекс внутри буфера (значение по умолчанию: 0);
- `length` — число байтов, доступных для записи в буфер (значение по умолчанию: размер буфера);
- `position` — позиция от начала файла, с которой начинается считывание, или значение `null` (значение по умолчанию: `null`).

В последнем параметре указывается функция, которая будет вызвана вне зависимости от успешности операции. Функция принимает три параметра. Через первый параметр доступен объект ошибки, если данные прочитать не удалось, или значение `null` — в случае успеха. Через второй параметр доступно число считанных байтов, а через третий — ссылка на буфер.

Если нужно прочитать файл целиком, можно воспользоваться функцией `readFile()` (см. *разд. 16.5*) или сначала получить размер файла с помощью функции `stat()`. С помощью кода, приведенного в листинге 16.27, прочитаем

все содержимое файла `cp1251.txt` (см. *разд. 16.3*) и преобразуем кодировку с помощью пакета `iconv-lite` (см. *разд. 15.14*).

Листинг 16.27. Чтение из файла функцией `read()`

```
const iconv = require('iconv-lite');
const fs = require('fs');
function closeFd(fd) {
  if (typeof fd === 'number') {
    fs.close(fd, function(err) {
      if (err) console.error(err.message);
      else console.log('Файл закрыт');
    });
  }
}
let p = 'cp1251.txt';
fs.stat(p, function(error, stats) {
  if (error) throw error;
  fs.open(p, 'r', function(err, fd) {
    if (err) throw err;
    console.log(`Файл успешно открыт. fd = ${fd}`);
    let buf = Buffer.alloc(stats.size);
    fs.read(fd, { buffer: buf, offset: 0, length: buf.length,
      position: null }, function(e, n, b) {
      if (e) {
        fs.closeSync(fd);
        throw e;
      }
      try {
        console.log(n);
        let str = iconv.decode(b, 'win1251');
        console.log(str);
      } finally {
        closeFd(fd);
      }
    });
  });
});
console.log('Конец программы');
```

Результат в консоли (мы успешно прочитали файл в кодировке Windows-1251):

Конец программы

Файл успешно открыт. fd = 3

15

Строка1

Строка2

Файл закрыт

- ❑ `readv()` — считывает данные из файла асинхронно и записывает их в несколько буферов. Формат функции:

```
readv(<Дескриптор>, <Массив>[, <Позиция>], <Функция>)
```

В параметре `<Массив>` указывается массив с объектами `Buffer`, `DataView` или типизированными массивами. Если указан параметр `<Позиция>`, то данные считываются, начиная с заданной позиции от начала файла. Если параметр не указан или тип параметра не является числом, то чтение выполняется, начиная с текущей позиции.

В последнем параметре указывается функция, которая будет вызвана вне зависимости от успешности операции. Функция принимает три параметра. Через первый параметр доступен объект ошибки, если данные прочитать не удалось, или значение `null` — в случае успеха. Через второй параметр доступно число считанных байтов, а через третий — ссылка на массив с объектами буферов.

С помощью кода, приведенного в листинге 16.28, прочитаем содержимое файла `file1.txt` (см. *разд. 16.3*).

Листинг 16.28. Чтение из файла функцией `readv()`

```
const fs = require('fs');
function closeFd(fd) {
  if (typeof fd === 'number') {
    fs.close(fd, function(err) {
      if (err) console.error(err.message);
      else console.log('Файл закрыт');
    });
  }
}
fs.open('file1.txt', 'r', function(err, fd) {
  if (err) throw err;
  console.log(`Файл успешно открыт. fd = ${fd}`);
  let buf1 = Buffer.alloc(15);
  let buf2 = Buffer.alloc(15);
  fs.readv(fd, [buf1, buf2], function(e, n, bs) {
    if (e) {
      fs.closeSync(fd);
      throw e;
    }
    try {
      console.log(n);
      console.log(bs);
    } finally {
      closeFd(fd);
    }
  });
});
console.log('Конец программы');
```

Результат в консоли:

Конец программы

Файл успешно открыт. fd = 3

27

```
[
  <Buffer d0 a1 d1 82 d1 80 d0 be d0 ba d0 b0 31 0a d0>,
  <Buffer a1 d1 82 d1 80 d0 be d0 ba d0 b0 32 00 00 00>
]
```

Файл закрыт

- `<FileHandle>.read()` — считывает данные из файла асинхронно. Формат метода:
- ```
<Promise> = <FileHandle>.read(<Buffer>, <Начало>, <Длина>, <Позиция>)
```

В качестве `<FileHandle>` указывается дескриптор файла, открытого с помощью функции `promises.open()`, а в параметре `<Buffer>` — буфер для хранения считанных данных в виде объекта `Buffer` или `Uint8Array`. В параметре `<Начало>` указывается начальный индекс внутри буфера, а в параметре `<Длина>` — число байтов, доступных для записи в буфер. В параметре `<Позиция>` задается позиция от начала файла, с которой начинается считывание. Если параметр имеет значение `null`, то считывание начинается с текущей позиции курсора и позиция курсора после завершения чтения будет обновлена. Если в параметре указано число, то позиция курсора не изменится.

Метод возвращает объект `Promise`, содержащий объект с двумя свойствами. Через свойство `bytesRead` доступно число считанных байтов, а через свойство `buffer` — заданный буфер.

Если нужно прочитать весь файл целиком, можно воспользоваться функцией `promises.readFile()` (см. *разд. 16.5*) или сначала получить размер файла с помощью метода `<FileHandle>.stat()`. С помощью кода, приведенного в листинге 16.29, прочитаем все содержимое файла `cp1251.txt` (см. *разд. 16.3*) и преобразуем кодировку с помощью пакета `iconv-lite` (см. *разд. 15.14*).

**Листинг 16.29. Чтение из файла методом `read()` 1**

```
const iconv = require('iconv-lite');
const fs = require('fs');
(async function() {
 let fh;
 try {
 fh = await fs.promises.open('cp1251.txt', 'r');
 console.log('Файл успешно открыт');
 let stats = await fh.stat();
 let buf = Buffer.alloc(stats.size);
 let obj = await fh.read(buf, 0, buf.length, null);
 console.log(obj);
 let str = iconv.decode(obj.buffer, 'win1251');
 console.log(str);
 }
})
```

```
 } catch (err) {
 console.error(err.message);
 } finally {
 if (fh !== undefined) {
 await fh.close();
 console.log('Файл закрыт');
 }
 }
 })();
console.log('Конец программы');
```

Результат в консоли (мы успешно прочитали файл в кодировке Windows-1251):

```
Конец программы
Файл успешно открыт
{
 bytesRead: 15,
 buffer: <Buffer d1 f2 f0 ee ea e0 31 0a d1 f2 f0 ee ea e0 32>
}
Строка1
Строка2
Файл закрыт
```

□ `<FileHandle>.read(<Опции>)` — аналогичен описанному ранее методу `<FileHandle>.read()`, только принимает единственный параметр, представляющий собой простой объект со следующими свойствами:

- `buffer` — буфер для хранения считанных данных в виде объекта `Buffer` или `Uint8Array`;
- `offset` — начальный индекс внутри буфера;
- `length` — число байтов, доступных для записи в буфер;
- `position` — позиция от начала файла, с которой начинается считывание. Если параметр имеет значение `null`, то считывание начинается с текущей позиции курсора и позиция курсора после завершения чтения будет обновлена. Если в параметре указано число, то позиция курсора не изменится.

С помощью кода, приведенного в листинге 16.30, прочитаем все содержимое файла `sr1251.txt` (см. *разд. 16.3*) и преобразуем кодировку с помощью пакета `iconv-lite` (см. *разд. 15.14*).

#### Листинг 16.30. Чтение из файла методом `read()` 2

```
const iconv = require('iconv-lite');
const fs = require('fs');
(async function() {
 let fh;
```



```

try {
 fh = await fs.promises.open('cp1251.txt', 'r');
 console.log('Файл успешно открыт');
 let stats = await fh.stat();
 let buf = Buffer.alloc(stats.size);
 const params = {
 buffer: buf,
 offset: 0,
 length: buf.length,
 position: null
 };
 let obj = await fh.read(params);
 console.log(obj);
 let str = iconv.decode(obj.buffer, 'win1251');
 console.log(str);
} catch (err) {
 console.error(err.message);
} finally {
 if (fh !== undefined) {
 await fh.close();
 console.log('Файл закрыт');
 }
}
})();
console.log('Конец программы');

```

**Результат в консоли (мы успешно прочитали файл в кодировке Windows-1251):**

```

Конец программы
Файл успешно открыт
{
 bytesRead: 15,
 buffer: <Buffer d1 f2 f0 ee ea e0 31 0a d1 f2 f0 ee ea e0 32>
}
Строка1
Строка2
Файл закрыт

```

- `<FileHandle>.readFile()` — считывает данные из файла, начиная с текущей позиции и до конца, асинхронно. Формат метода:

```
<Promise> = <FileHandle>.readFile([<Опции>])
```

В качестве `<FileHandle>` указывается дескриптор файла, открытого с помощью функции `promises.open()`. В параметре `<Опции>` можно указать кодировку в виде строки или объект со свойством `encoding` (значение по умолчанию: `null`), задающим кодировку. Если свойство `encoding` имеет значение `null`, метод вернет объект `Promise`, содержащий объект `Buffer`. Если же кодировка указана, метод вернет объект `Promise`, содержащий строку.

С помощью кода, приведенного в листинге 16.31, прочитаем все содержимое файла `cp1251.txt` (см. *разд. 16.3*) и преобразуем кодировку с помощью пакета `iconv-lite` (см. *разд. 15.14*).

**Листинг 16.31. Чтение из файла методом `readFile()`**

```
const iconv = require('iconv-lite');
const fs = require('fs');
(async function() {
 let fh;
 try {
 fh = await fs.promises.open('cp1251.txt', 'r');
 console.log('Файл успешно открыт');
 let buf = await fh.readFile({ encoding: null });
 let str = iconv.decode(buf, 'win1251');
 console.log(str);
 } catch (err) {
 console.error(err.message);
 } finally {
 if (fh !== undefined) {
 await fh.close();
 console.log('Файл закрыт');
 }
 }
})();
console.log('Конец программы');
```

Результат в консоли (мы успешно прочитали файл в кодировке Windows-1251):

```
Конец программы
Файл успешно открыт
Строка1
Строка2
Файл закрыт
```

- `<FileHandle>.readv()` — считывает данные из файла асинхронно и записывает их в несколько буферов. Формат метода:

```
<Promise> = <FileHandle>.readv(<Массив>[, <Позиция>])
```

В качестве `<FileHandle>` указывается дескриптор файла, открытого с помощью функции `promises.open()`. В параметре `<Массив>` задается массив с объектами `Buffer`, `DataView` или типизированными массивами. Если указан параметр `<Позиция>`, то данные считываются, начиная с заданной позиции от начала файла. Если тип параметра не является числом, то чтение выполняется, начиная с текущей позиции.

Метод возвращает объект `Promise`, содержащий объект с двумя свойствами. Через свойство `bytesRead` доступно число считанных байтов, а через свойство `buffers` — массив с объектами буферов.

С помощью кода, приведенного в листинге 16.32, прочитаем содержимое файла `file1.txt` (см. *разд. 16.3*).

**Листинг 16.32. Чтение из файла методом `readv()`**

```
const fs = require('fs');
(async function() {
 let fh;
 try {
 fh = await fs.promises.open('file1.txt', 'r');
 console.log('Файл успешно открыт');
 let buf1 = Buffer.alloc(15);
 let buf2 = Buffer.alloc(15);
 let obj = await fh.readv([buf1, buf2]);
 console.log(obj);
 } catch (err) {
 console.error(err.message);
 } finally {
 if (fh !== undefined) {
 await fh.close();
 console.log('Файл закрыт');
 }
 }
})();
console.log('Конец программы');
```

**Результат в консоли:**

```
Конец программы
Файл успешно открыт
{
 bytesRead: 27,
 buffers: [
 <Buffer d0 a1 d1 82 d1 80 d0 be d0 ba d0 b0 31 0a d0>,
 <Buffer a1 d1 82 d1 80 d0 be d0 ba d0 b0 32 00 00 00>
]
}
Файл закрыт
```

## 16.9. Изменение размера файла

Изменить размер файла позволяют следующие функции из модуля `fs`:

❑ `truncateSync()` — синхронное изменение размера. Формат функции:

```
truncateSync(<Путь к файлу>[, <Размер>])
```

В первом параметре указывается путь к файлу в виде строки, объекта `Buffer` или `URL`. Если указан относительный путь, он разрешается относительно текущего рабочего каталога (см. *разд. 16.1*).

В параметре `<Размер>` указывается новый размер файла. Если параметр не указан, то по умолчанию используется значение 0. Если размер файла больше указанного значения, то файл обрезается, а если меньше — то в файл добавляются нулевые байты, чтобы его размер был равен заданному.

Пример приведен в листинге 16.33.

**Листинг 16.33. Изменение размера файла функцией `truncateSync()`**

```
const fs = require('fs');
let p = 'file2.txt';
try {
 console.log(fs.statSync(p).size); // 27
 fs.truncateSync(p);
 console.log(fs.statSync(p).size); // 0
 fs.truncateSync(p, 5);
 console.log(fs.statSync(p).size); // 5
} catch (e) {
 console.log(e.message);
}
```

- `ftruncateSync()` — синхронное изменение размера открытого файла. Формат функции:

```
ftruncateSync(<Дескриптор>[, <Размер>])
```

В первом параметре указывается дескриптор файла. В параметре `<Размер>` задается новый размер файла. Если параметр не указан, то по умолчанию используется значение 0. Если размер файла больше указанного значения, то файл обрезается, а если меньше — то в файл добавляются нулевые байты, чтобы его размер стал равен заданному.

Пример приведен в листинге 16.34.

**Листинг 16.34. Изменение размера файла функцией `ftruncateSync()`**

```
const fs = require('fs');
let p = 'file2.txt';
let fd;
try {
 console.log(fs.statSync(p).size); // 5
 fd = fs.openSync(p, 'r+');
 fs.ftruncateSync(fd, 10);
 fs.closeSync(fd);
 fd = undefined;
 console.log(fs.statSync(p).size); // 10
}
```

```
 } finally {
 if (fd !== undefined) fs.closeSync(fd);
 }
 }
```

- ❑ `truncate()` — асинхронное изменение размера. Формат функции:

`truncate(<Путь к файлу>[, <Размер>], <Функция>)`

В первом параметре указывается путь к файлу в виде строки, объекта `Buffer` или URL. Если указан относительный путь, он разрешается относительно текущего рабочего каталога (см. *разд. 16.1*).

В параметре `<Размер>` задается новый размер файла. Если параметр не указан, то по умолчанию используется значение 0. Если размер файла больше указанного значения, то файл обрезается, а если меньше — то в файл добавляются нулевые байты, чтобы его размер стал равным заданному.

В последнем параметре указывается функция, которая будет вызвана вне зависимости от успешности операции. Функция принимает один параметр, через который доступен объект ошибки, если не удалось изменить размер файла, или значение `null` — если операция выполнена успешно.

Пример приведен в листинге 16.35.

**Листинг 16.35. Изменение размера файла функцией `truncate()`**

```
const fs = require('fs');
let p = 'file2.txt';
console.log(fs.statSync(p).size); // 10
fs.truncate(p, 5, function(err) {
 if (err) throw err;
 console.log(fs.statSync(p).size); // 5
});
```

- ❑ `ftruncate()` — асинхронное изменение размера открытого файла. Формат функции:

`ftruncate(<Дескриптор>[, <Размер>], <Функция>)`

В первом параметре указывается дескриптор файла. В параметре `<Размер>` задается новый размер файла. Если параметр не указан, то по умолчанию используется значение 0. Если размер файла больше указанного значения, то файл обрезается, а если меньше — то в файл добавляются нулевые байты, чтобы его размер стал равным заданному.

В последнем параметре указывается функция, которая будет вызвана вне зависимости от успешности операции. Функция принимает один параметр, через который доступен объект ошибки, если не удалось изменить размер файла, или значение `null` — если операция выполнена успешно.

Пример приведен в листинге 16.36.

**Листинг 16.36. Изменение размера файла функцией `ftruncate()`**

```
const fs = require('fs');
let p = 'file2.txt';
console.log(fs.statSync(p).size); // 5
fs.open(p, 'r+', function(err, fd) {
 if (err) throw err;
 fs.ftruncate(fd, 10, function(err) {
 fs.closeSync(fd);
 if (err) throw err;
 console.log(fs.statSync(p).size); // 10
 });
});
```

- `promises.truncate()` — асинхронное изменение размера. Функция возвращает объект `Promise` (см. *разд. 14.1*). Формат функции:

```
<Promise> = promises.truncate(<Путь к файлу>[, <Размер>])
```

В первом параметре указывается путь к файлу в виде строки, объекта `Buffer` или `URL`. Если указан относительный путь, он разрешается относительно текущего рабочего каталога (см. *разд. 16.1*).

В параметре `<Размер>` задается новый размер файла. Если параметр не указан, то по умолчанию используется значение 0. Если размер файла больше указанного значения, то файл обрезается, а если меньше — то в файл добавляются нулевые байты, чтобы его размер стал равным заданному.

Пример приведен в листинге 16.37.

**Листинг 16.37. Изменение размера файла методом `truncate()` 1**

```
const fs = require('fs');
(async function() {
 try {
 let p = 'file2.txt';
 let stats = await fs.promises.stat(p);
 console.log(stats.size); // 10
 await fs.promises.truncate(p, 5);
 stats = await fs.promises.stat(p);
 console.log(stats.size); // 5
 } catch (err) {
 console.error(err.message);
 }
})();
```

- `<FileHandle>.truncate()` — асинхронное изменение размера открытого файла. Метод возвращает объект `Promise` (см. *разд. 14.1*). Формат метода:

```
<Promise> = <FileHandle>.truncate(<Размер>)
```

В качестве `<FileHandle>` указывается дескриптор файла, открытого с помощью функции `promises.open()`. В параметре `<Размер>` задается новый размер файла. Если размер файла больше указанного значения, то файл обрезается, а если меньше — то в файл добавляются нулевые байты, чтобы его размер стал равным заданному.

Пример приведен в листинге 16.38.

#### Листинг 16.38. Изменение размера файла методом `truncate()` 2

```
const fs = require('fs');
(async function() {
 let fh;
 let p = 'file2.txt';
 try {
 let stats = await fs.promises.stat(p);
 console.log(stats.size); // 5
 fh = await fs.promises.open(p, 'r+');
 await fh.truncate(10);
 await fh.close();
 fh = undefined;
 stats = await fs.promises.stat(p);
 console.log(stats.size); // 10
 } finally {
 if (fh !== undefined) await fh.close();
 }
})();
```

## 16.10. Дескрипторы стандартных потоков ввода/вывода

Потоки ввода/вывода в консоль аналогичны потокам ввода/вывода в файл. При запуске программы автоматически открываются три потока:

- ❑ `stdin` — стандартный ввод;
- ❑ `stdout` — стандартный вывод;
- ❑ `stderr` — стандартный вывод сообщений об ошибках.

Получить доступ к объектам потоков позволяет модуль `process`. С помощью свойства `fd` можно получить файловый дескриптор и использовать его в функциях, предназначенных для работы с файлами. Выведем дескрипторы стандартных потоков:

```
console.log(process.stdin.fd); // 0
console.log(process.stdout.fd); // 1
console.log(process.stderr.fd); // 2
```

Например, вывести строку в консоль можно так:

```
const fs = require('fs');
fs.writeFileSync(process.stdout.fd, 'Строка');
```

Однако тут существует проблема. Если запустить этот код, то прочитать сообщение мы не сможем из-за неправильной кодировки. Консоль работает с кодировкой Windows-866, а строка у нас в кодировке UTF-8. Конечно, можно сменить кодировку консоли с помощью команды `chcp 65001`:

```
C:\book\pl>chcp 65001
Active code page: 65001
```

По умолчанию используются точечные шрифты, поэтому русские символы опять будут искажены. Однако можно зайти в свойства консоли и выбрать шрифт **Lucida Console**, тогда текст будет отображаться правильно:

```
C:\book\pl>node index.js
Строка
```

Проблема в том, что из программы сменить шрифт консоли мы не можем. Вернем кодировку по умолчанию:

```
C:\book\pl>chcp 866
Текущая кодовая страница: 866
```

Чтобы записать в поток данные в кодировке Windows-866, следует воспользоваться пакетом `iconv-lite` (см. *разд. 15.14*):

```
const fs = require('fs');
const iconv = require('iconv-lite');
let buf = iconv.encode('Строка', 'cp866');
fs.writeFileSync(process.stdout.fd, buf);
```

Однако пользователь может сменить кодировку в консоли, и мы опять получим проблему с русскими буквами. Как выход, можно выводить сообщения на английском языке. Коды латинских букв во всех однобайтовых кодировках одинаковые, поэтому при использовании английского языка проблем не будет. Но наилучшее решение проблемы — применение для вывода методов объекта `console`, которые преобразуют кодировки автоматически.





## ГЛАВА 17

# Файловые потоки ВВОДА/ВЫВОДА

Для чтения и записи файлов в Node.js можно использовать *потоки*, а именно *поток ввода* (объект для чтения данных из файла) и *поток вывода* (объект для записи данных в файл). Потоки не загружают обрабатываемый файл целиком в память, что позволяет сэкономить системные ресурсы при обработке файлов большого размера. Потоки могут работать не только с файлами, но и с другими источниками данных, например консолью.

### 17.1. Класс *WriteStream*: поток вывода в файл

Класс `WriteStream` реализует поток вывода, с помощью которого можно записывать данные в файл.

#### 17.1.1. Создание потока вывода

Создать поток вывода позволяет функция `createWriteStream()` из модуля `fs`. Формат функции:

```
createWriteStream(<Путь к файлу>[, <Опции или кодировка>])
```

В первом параметре указывается путь к файлу в виде строки, объекта `Buffer` или URL. Если указан относительный путь, он разрешается относительно текущего рабочего каталога (см. *разд. 16.1*). Функция возвращает ссылку на поток вывода `WriteStream`.

Запишем две строки в файл в кодировке UTF-8 (если файл не существует, он будет создан, а если существует — перезаписан):

```
const fs = require('fs');
const path = require('path');
let p = path.join(__dirname, 'test.txt');
let stream = fs.createWriteStream(p);
stream.on('error', err => console.log('Ошибка:', err));
stream.on('finish', () => console.log('Данные успешно записаны'));
```

```
stream.write('Строка1\n');
stream.end('Строка2');
console.log('Конец программы');
```

Результат в консоли при успешном выполнении операции (обратите внимание: обработчики вызываются асинхронно):

```
Конец программы
Данные успешно записаны
```

Во втором параметре можно указать строку с кодировкой или объект со следующими основными свойствами (полный список приведен в документации):

- ❑ `encoding` — кодировка в виде строки (значение по умолчанию: `'utf8'`);
- ❑ `mode` — режим доступа к файлу в виде восьмеричного числа (значение по умолчанию: `0o666`);
- ❑ `flags` — режим открытия файла в виде строки (значение по умолчанию: `'w'`; если файл не существует, он будет создан, а если существует — перезаписан). Режимы открытия файла рассматривались в *разд. 16.6*;
- ❑ `fd` — дескриптор файла, с которым будет работать создаваемый поток (значение по умолчанию: `null`). Если свойству присвоен дескриптор, то путь, указанный в первом параметре, игнорируется;
- ❑ `start` — начальная позиция записи в файле. Режим открытия файла должен позволять записывать данные с указанной позиции;
- ❑ `autoClose` — если `true` (значение по умолчанию), то файл будет автоматически закрыт после записи;
- ❑ `emitClose` — если `true` (значение по умолчанию), то при закрытии файла будет сгенерировано событие `close`.

Добавим две строки в конец файла (листинг 17.1).

#### Листинг 17.1. Добавление строк в файл с использованием потоков

```
const fs = require('fs');
const path = require('path');
let p = path.join(__dirname, 'test.txt');
let stream = fs.createWriteStream(p, {
 flags: 'a',
 encoding: 'utf8'
});
stream.on('error', err => console.log('Ошибка:', err));
stream.on('finish', () => console.log('Данные успешно записаны'));
stream.write('\nСтрока3\n');
stream.end('Строка4');
```

Класс `WriteStream` содержит свойства `bytesWritten` (число записанных байтов) и `path` (путь к обрабатываемому файлу).

## 17.1.2. Запись данных

Свойство `writable` объекта потока хранит значение `true`, если в поток можно записывать данные, и `false`, если поток не предназначен для записи:

```
if (stream.writable) {
 // Можно записывать данные
}
```

Записать данные в поток позволяют следующие методы:

- `write()` — записывает данные в поток вывода. Формат метода:

```
write(<Данные>[, <Кодировка>][, <Функция>])
```

Записываемые данные указываются в виде строки, объекта `Buffer` или `Uint8Array`. Второй параметр позволяет задать кодировку (значение по умолчанию: `'utf8'`). В последнем параметре можно передать ссылку на функцию, вызываемую после успешной записи данных.

Метод возвращает значение `true` в случае успешной записи данных, и `false` — если записать данные в текущий момент невозможно. В последнем случае следует приостановить запись данных, пока не будет сгенерировано событие `drain` (см. *разд. 17.1.3*);

- `end()` — записывает данные в поток вывода и завершает работу с потоком. Вызов метода `write()` после выполнения метода `end()` вызовет ошибку. Формат метода:

```
end([<Данные>[, <Кодировка>]][, <Функция>])
```

Параметры аналогичны таковым у метода `write()`. В качестве результата возвращается `this`.

Пример записи данных в кодировке `Windows-1251` (установка пакета `iconv-lite` описана в *разд. 15.14*) приведен в листинге 17.2.

### Листинг 17.2. Запись данных в кодировке `Windows-1251` в поток

```
const fs = require('fs');
const iconv = require('iconv-lite');
const path = require('path');
let p = path.join(__dirname, 'cp1251.txt');
let stream = fs.createWriteStream(p, { encoding: null });
stream.on('error', err => console.log('Ошибка:', err));
stream.on('finish', () => console.log('Данные успешно записаны'));
let data = iconv.encode('Строка1', 'win1251');
stream.write(data);
stream.end();
```

Чтобы проверить, был ли уже вызван метод `end()` и, соответственно, можно ли еще выполнять запись в потоке, достаточно обратиться к свойству `writableEnded`

объекта потока. Оно хранит значение `true`, если метод `end()` уже был вызван (и соответственно запись в поток больше невозможна), и `false` — в противном случае. Пример:

```
if (!stream.writableEnded) {
 // Записывать данные в поток еще можно
} else {
 // Уже нельзя!
}
```

### 17.1.3. События потока вывода

Отслеживать процесс записи данных можно внутри обработчиков следующих событий (полный список событий приведен в документации):

- ❑ `open` — генерируется при открытии файла. Внутри обработчика через параметр доступен целочисленный дескриптор файла;
- ❑ `ready` — поток готов к работе. Генерируется после события `open`;
- ❑ `finish` — генерируется после вызова метода `end()`;
- ❑ `close` — файл закрыт;
- ❑ `drain` — генерируется, когда вновь появляется возможность записать данные в поток;
- ❑ `error` — возникла ошибка. Внутри обработчика через параметр доступен объект ошибки.

Пример обработки событий приведен в листинге 17.3.

#### Листинг 17.3. Обработка событий потока вывода

```
const fs = require('fs');
const path = require('path');
let p = path.join(__dirname, 'test.txt');
let stream = fs.createWriteStream(p, { emitClose: true });
stream.on('open', fd => {
 console.log('Событие open. fd=', fd);
});
stream.on('ready', () => console.log('Событие ready'));
stream.on('finish', () => {
 console.log('Событие finish. Записано:', stream.bytesWritten);
});
stream.on('close', () => console.log('Событие close'));
stream.on('drain', () => console.log('Событие drain'));
stream.on('error', err => console.log('Ошибка:', err));
for (let i = 1; i <= 100; i++) {
 stream.write(`Строка${i}\n`);
}
stream.end();
```

Результат в консоли при успешной записи:

```
Событие open. fd= 3
Событие ready
Событие finish. Записано: 1492
Событие close
```

## 17.2. Класс *ReadStream*: поток ввода из файла

Класс `ReadStream` реализует поток ввода, с помощью которого можно считывать данные из файла. При чтении используется буфер.

### 17.2.1. Создание потока ввода

Создать поток ввода позволяет функция `createReadStream()` из модуля `fs`. Формат функции:

```
createReadStream(<Путь к файлу>[, <Опции или кодировка>])
```

В первом параметре указывается путь к файлу в виде строки, объекта `Buffer` или URL. Если указан относительный путь, он разрешается относительно текущего рабочего каталога (см. *разд. 16.1*). Функция возвращает ссылку на поток ввода `ReadStream`.

Прочитаем файл в кодировке UTF-8 (обратите внимание: обработчики вызываются асинхронно):

```
const fs = require('fs');
const path = require('path');
let p = path.join(__dirname, 'test.txt');
let stream = fs.createReadStream(p);
stream.setEncoding('utf8');
stream.on('data', chunk => {
 console.log(chunk);
});
stream.on('error', err => console.log('Ошибка:', err));
```

Во втором параметре можно указать строку с кодировкой или объект со следующими основными свойствами (полный список приведен в документации):

- ❑ `encoding` — кодировка в виде строки (значение по умолчанию: `null`);
- ❑ `mode` — режим доступа к файлу в виде восьмеричного числа (значение по умолчанию: `0o666`);
- ❑ `flags` — режим открытия файла в виде строки (значение по умолчанию: `'r'`; только чтение). Режимы открытия файла рассматривались в *разд. 16.6*;
- ❑ `fd` — дескриптор файла, с которым будет работать создаваемый поток (значение по умолчанию: `null`). Если свойству присвоен дескриптор, то путь, указанный в первом параметре, игнорируется;

- ❑ `highWaterMark` — объем внутреннего буфера, в котором временно сохраняются прочитанные из файла данные (значение по умолчанию:  $64 \cdot 1024 = 64$  Кбайт);
- ❑ `start` — начальная позиция считывания;
- ❑ `end` — конечная позиция считывания (значение по умолчанию: `Infinity`);
- ❑ `autoClose` — если `true` (значение по умолчанию), файл будет автоматически закрыт после чтения;
- ❑ `emitClose` — если `true` (значение по умолчанию), при закрытии файла будет сгенерировано событие `close`.

## 17.2.2. Чтение данных

Свойство `readable` объекта потока хранит значение `true`, если из потока можно читать данные, и `false`, если поток не предназначен для чтения:

```
if (stream.readable) {
 // Можно читать данные
}
```

Для чтения данных нужно назначить обработчик для события `data`. Внутри обработчика через параметр доступны данные. Событие будет генерироваться несколько раз, т. к. чтение выполняется отдельными частями, записываемыми во внутренний буфер. Если кодировка не задана (значение по умолчанию), данные будут иметь тип `Buffer`. Задать кодировку можно с помощью второго параметра функции `createReadStream()` или метода `setEncoding(<Кодировка>)`. Пример указания свойства `encoding`:

```
const fs = require('fs');
const path = require('path');
let p = path.join(__dirname, 'test.txt');
let stream = fs.createReadStream(p, { encoding: 'utf8' });
stream.on('data', chunk => {
 console.log(chunk);
});
stream.on('error', err => console.log('Ошибка:', err));
```

Также можно назначить обработчик для события `readable` и внутри обработчика использовать метод `read([<Размер>])`. Если параметр не задан, возвращаются все данные из буфера в виде строки или объекта `Buffer`. Если данных больше нет, метод вернет значение `null`. Получим данные в виде объекта `Buffer`:

```
const fs = require('fs');
const path = require('path');
let p = path.join(__dirname, 'test.txt');
let stream = fs.createReadStream(p);
stream.on('readable', () => {
 let chunk;
```

```
while (null !== (chunk = stream.read())) {
 console.log(chunk);
}
});
stream.on('error', err => console.log('Ошибка:', err));
```

### 17.2.3. События потока ввода

Поток ввода поддерживает следующие события (полный список событий приведен в документации):

- ❑ `open` — генерируется при открытии файла. Внутри обработчика через параметр доступен целочисленный дескриптор файла;
- ❑ `ready` — поток готов к работе. Генерируется после события `open`;
- ❑ `readable` — либо данные считаны из файла и готовы к получению с помощью метода `read()`, либо достигнут конец файла (в последнем случае генерируется перед событием `end`);
- ❑ `data` — данные считаны из файла и готовы к получению внутри обработчика через параметр. Если кодировка не задана, данные будут иметь тип `Buffer`;
- ❑ `end` — достигнут конец файла;
- ❑ `close` — файл закрыт;
- ❑ `error` — генерируется при ошибке. Внутри обработчика через параметр доступен объект ошибки.

Пример обработки событий приведен в листинге 17.4.

#### Листинг 17.4. Обработка событий потока ввода

```
const fs = require('fs');
const path = require('path');
let p = path.join(__dirname, 'test.txt');
let stream = fs.createReadStream(p, { emitClose: true });
stream.on('open', fd => {
 console.log('Событие open. fd=', fd);
});
stream.on('ready', () => console.log('Событие ready'));
stream.on('data', chunk => {
 console.log('Событие data');
});
stream.on('readable', () => {
 console.log('Событие readable');
 let chunk = stream.read();
});
stream.on('end', () => console.log('Событие end'));
stream.on('close', () => console.log('Событие close'));
stream.on('error', err => console.log('Ошибка:', err));
```

Результат в консоли:

```
Событие open. fd= 3
Событие ready
Событие readable
Событие data
Событие readable
Событие end
Событие close
```

## 17.2.4. Метод *pipe()*

Метод `pipe()` позволяет присоединить поток вывода к потоку ввода. При этом данные после чтения из файла сразу будут передаваться потоку вывода для записи в другой файл. Метод вызывается у потока ввода. Формат метода:

```
pipe(<Поток вывода>[, <Опции>])
```

Во втором параметре указывается объект со свойством `end`. Если свойство имеет значение `true` (значение по умолчанию), то для потока будет вызван метод `end()`. Метод возвращает ссылку на поток.

Когда у потока ввода вызывается метод `pipe()`, в потоке вывода, указанного в качестве первого параметра, генерируется событие `pipe`. Его обработчику в качестве параметра передается ссылка на текущий поток ввода.

Пример создания архива приведен в листинге 17.5.

### Листинг 17.5. Создание архива

```
const fs = require('fs');
const zlib = require('zlib');
const path = require('path');
let p = path.join(__dirname, 'test.txt');
let p2 = path.join(__dirname, 'test.txt.gz');
let rstream = fs.createReadStream(p);
let wstream = fs.createWriteStream(p2);
wstream.on('pipe', src => console.log('Потоки соединены'));
rstream.on('error', err => console.log('Ошибка:', err));
rstream.on('end', () => console.log('Операция выполнена'));
let zstream = zlib.createGzip();
rstream.pipe(zstream).pipe(wstream, { end: true });
```

Для отсоединения потока предназначен метод `unpipe(<Поток вывода>)`, также вызываемый у потока ввода. При вызове этого метода в отсоединяемом потоке вывода генерируется событие `unpipe`. Его обработчику в качестве параметра передается ссылка на текущий поток ввода.



### 17.2.5. Чтение файла, сохраненного в русской кодировке

Прочитаем файл в кодировке Windows-1251 и запишем данные в файл в кодировке UTF-8 (листинг 17.6), применив пакет `iconv-lite` (см. *разд. 15.14*).

#### Листинг 17.6. Преобразование файла из кодировки Windows-1251 в UTF-8

```
const fs = require('fs');
const iconv = require('iconv-lite');
const path = require('path');
let p = path.join(__dirname, 'cp1251.txt');
let p2 = path.join(__dirname, 'utf8.txt');
let rstream = fs.createReadStream(p);
let wstream = fs.createWriteStream(p2, { encoding: null });
rstream.on('error', err => console.log('Ошибка:', err));
rstream.on('end', () => console.log('Операция выполнена'));
rstream.pipe(iconv.decodeStream('win1251'))
 .pipe(iconv.encodeStream('utf8'))
 .pipe(wstream);
```

Прочитаем файл в кодировке Windows-866 и выведем данные в консоль (листинг 17.7).

#### Листинг 17.7. Чтение файла в кодировке Windows-866

```
const fs = require('fs');
const iconv = require('iconv-lite');
const path = require('path');
let p = path.join(__dirname, 'cp866.txt');
let rstream = fs.createReadStream(p);
let dstream = iconv.decodeStream('cp866');
rstream.on('error', err => console.log('Ошибка:', err));
dstream.on('data', chunk => console.log(chunk));
rstream.pipe(dstream);
```

## 17.3. Перенаправление стандартных потоков ввода/вывода

Стандартные потоки можно перенаправить таким образом, чтобы данные записывались в файл или считывались из файла. Чтобы выполнить перенаправление вывода в файл, в командной строке следует выполнить одну из следующих команд:

```
node index.js > file.txt
node index.js >> file.txt
```

Первая строка записывает результат выполнения программы `index.js` в файл `file.txt`. Если файл не существует, он будет создан, а если существует — перезаписан. Вторая строка выполняет дозапись в конец указанного файла.

Предыдущий пример сохранял в файл только данные из потока `stdout`. Чтобы сохранить данные из потока `stderr`, нужно дополнительно указать дескриптор потока. Поток `stdout` имеет дескриптор 1, а поток `stderr` — дескриптор 2:

```
node index.js 1> путь_к_файлу_для_out 2> путь_к_файлу_для_err
node index.js 1>> путь_к_файлу_для_out 2>> путь_к_файлу_для_err
```

Пример:

```
node index.js 1> out.txt 2> err.txt
```

Чтобы ввести в программу данные из файла, следует выполнить команду:

```
node index.js < file.txt
```



## ГЛАВА 18

# Работа с файловой системой

Переходим к рассмотрению средств для переименования, удаления файлов, получения информации о файлах, создания, удаления и чтения каталогов. Почти все функции, рассматриваемые в этой главе, расположены во встроенном модуле `fs`. Не забудьте при использовании модуля CommonJS в начало программы добавить следующую инструкцию:

```
const fs = require('fs');
```

Внутри модуля ECMAScript инструкция выглядит так:

```
import * as fs from 'fs';
```

## 18.1. Переименование и перемещение файлов

Для переименования и перемещения файлов предназначены следующие функции из модуля `fs`:

- ❑ `renameSync(<Старый путь>, <Новый путь>)` — переименовывает файл или каталог в синхронном режиме. В первом параметре указывается путь (в виде строки, объекта `Buffer` или `URL`) к файлу (каталогу), который требуется переименовать. Если файл (каталог) не существует, генерируется исключение. Во втором параметре указывается новый путь (в виде строки, объекта `Buffer` или `URL`). Если файл по новому пути существует, он будет перезаписан. Если каталог по новому пути существует, будет сгенерировано исключение. Пример переименования файла приведен в листинге 18.1.

### Листинг 18.1. Переименование файла функцией `renameSync()`

```
const fs = require('fs');
const path = require('path');
try {
 let oldPath = path.join(__dirname, 'file2.txt');
 let newPath = path.join(__dirname, 'file_2.txt');
```

```
fs.renameSync(oldPath, newPath);
console.log('Файл успешно переименован');
} catch (e) {
 console.error(e.message);
}
console.log('Конец программы');
```

Результат в консоли (обратите внимание: инструкции выполняются последовательно, т. е. синхронно):

```
Файл успешно переименован
Конец программы
```

Если новый путь не совпадает со старым, файл будет перемещен.

Каталоги можно только переименовать, перемещать их нельзя. Создадим каталог `folder1` в каталоге `C:\book\r1`, чтобы избежать ошибки, а затем переместим в него файл `file_2.txt` (листинг 18.2).

#### Листинг 18.2. Перемещение файла функцией `renameSync()`

```
const fs = require('fs');
const path = require('path');
try {
 let oldPath = path.join(__dirname, 'file_2.txt');
 let newPath = path.join(__dirname, 'folder1', 'file_2.txt');
 fs.renameSync(oldPath, newPath);
 console.log('Файл успешно перемещен');
} catch (e) {
 console.error(e.message);
}
```

- ❑ `rename(<Старый путь>, <Новый путь>, <Функция>)` — переименовывает файл или каталог в асинхронном режиме. В первом параметре указывается путь (в виде строки, объекта `Buffer` или `URL`) к файлу (каталогу), который требуется переименовать. Если файл (каталог) не существует, операция заканчивается ошибкой. Во втором параметре указывается новый путь (в виде строки, объекта `Buffer` или `URL`). Если файл по новому пути существует, он будет перезаписан. Если каталог по новому пути существует, операция заканчивается ошибкой. В последнем параметре указывается функция, которая будет вызвана вне зависимости от успешности операции. Функция принимает один параметр, через который доступен объект ошибки, если переименовать файл (каталог) не удалось, или значение `null` — в случае успеха. Пример переименования файла приведен в листинге 18.3.

#### Листинг 18.3. Переименование файла функцией `rename()`

```
const fs = require('fs');
const path = require('path');
```

```

let oldPath = path.join(__dirname, 'file3.txt');
let newPath = path.join(__dirname, 'file_3.txt');
fs.rename(oldPath, newPath, function(err) {
 if (err) {
 console.error(e.message);
 return;
 }
 console.log('Файл успешно переименован');
});
console.log('Конец программы');

```

Результат в консоли (обратите внимание: инструкции выполняются асинхронно):

```

Конец программы
Файл успешно переименован

```

Если новый путь не совпадает со старым, файл будет перемещен.

Как уже упоминалось, каталоги можно только переименовать, перемещать их нельзя. Переместим файл `file_3.txt` в каталог `folder1` (листинг 18.4).

#### Листинг 18.4. Перемещение файла функцией `rename()`

```

const fs = require('fs');
const path = require('path');
let oldPath = path.join(__dirname, 'file_3.txt');
let newPath = path.join(__dirname, 'folder1', 'file_3.txt');
fs.rename(oldPath, newPath, function(err) {
 if (err) {
 console.error(e.message);
 return;
 }
 console.log('Файл успешно перемещен');
});

```

- `promises.rename(<Старый путь>, <Новый путь>)` — переименовывает файл или каталог в асинхронном режиме. В первом параметре указывается путь (в виде строки, объекта `Buffer` или `URL`) к файлу (каталогу), который требуется переименовать. Если файл (каталог) не существует, операция заканчивается ошибкой. Во втором параметре указывается новый путь (в виде строки, объекта `Buffer` или `URL`). Если файл по новому пути существует, он будет перезаписан. Если каталог по новому пути существует, операция заканчивается ошибкой. Функция возвращает объект `Promise` (см. *разд. 14.1*). Пример переименования файла приведен в листинге 18.5.

#### Листинг 18.5. Переименование файла методом `rename()`

```

const fs = require('fs');
const path = require('path');

```

```
let oldPath = path.join(__dirname, 'file4.txt');
let newPath = path.join(__dirname, 'file_4.txt');
(async function() {
 try {
 await fs.promises.rename(oldPath, newPath);
 console.log('Файл успешно переименован');
 } catch (err) {
 console.error(err.message);
 }
})();
console.log('Конец программы');
```

**Результат в консоли (обратите внимание: инструкции выполняются асинхронно):**

```
Конец программы
Файл успешно переименован
```

Если новый путь не совпадает со старым, файл будет перемещен.

Переместим файл `file_4.txt` в каталог `folder1` (листинг 18.6).

#### Листинг 18.6. Перемещение файла методом `rename()`

```
const fs = require('fs');
const path = require('path');
let oldPath = path.join(__dirname, 'file_4.txt');
let newPath = path.join(__dirname, 'folder1', 'file_4.txt');
(async function() {
 try {
 await fs.promises.rename(oldPath, newPath);
 console.log('Файл успешно перемещен');
 } catch (err) {
 console.error(err.message);
 }
})();
```

## 18.2. Копирование файлов

Копировать файлы позволяют следующие функции из модуля `fs`:

❑ `copyFileSync()` — синхронное копирование. Формат функции:

```
copyFileSync(<Что копируем>, <Куда копируем>[, <Режим>])
```

В первом параметре указывается путь (в виде строки, объекта `Buffer` или `URL`) к файлу, который требуется скопировать. Если файл не существует, генерируется исключение. Во втором параметре указывается путь назначения (в виде строки, объекта `Buffer` или `URL`). Если файл с указанным путем назначения существует, то по умолчанию он будет перезаписан. Чтобы в этом случае генерировалось ис-

ключение, следует в третьем параметре передать флаг `constants.COPYFILE_EXCL`. В параметре `<Режим>` можно указать комбинацию следующих констант через оператор `|`:

- `COPYFILE_EXCL` — если файл с указанным путем назначения существует, не перезаписывать его, а сгенерировать исключение;
- `COPYFILE_FICLONE` — по возможности использовать функциональность создания копии файла по запросу, при которой копия файла реально создается лишь при первой попытке записать в нее данные. Если текущая платформа не поддерживает такую функциональность, создается обычная копия файла;
- `COPYFILE_FICLONE_FORCE` — всегда использовать функциональность создания копии файла по запросу. Если текущая платформа не поддерживает ее, генерируется исключение.

Пример копирования файла приведен в листинге 18.7.

#### Листинг 18.7. Копирование файла функцией `copyFileSync()`

```
const fs = require('fs');
const path = require('path');
let srcPath = path.join(__dirname, 'file1.txt');
let destPath = path.join(__dirname, 'folder1', 'file_1.txt');
try {
 fs.copyFileSync(srcPath, destPath,
 fs.constants.COPYFILE_EXCL);
 console.log('Файл успешно скопирован');
} catch (err) {
 console.error(err.message);
}
console.log('Конец программы');
```

Результат в консоли (обратите внимание: выполнение инструкций происходит последовательно, т. е. синхронно):

```
Файл успешно скопирован
Конец программы
```

#### ❑ `copyFile()` — асинхронное копирование. Формат функции:

```
copyFile(<Что копируем>, <Куда копируем>[, <Режим>], <Функция>)
```

В первом параметре указывается путь (в виде строки, объекта `Buffer` или `URL`) к файлу, который требуется скопировать. Если файл не существует, операция заканчивается ошибкой. Во втором параметре указывается путь назначения (в виде строки, объекта `Buffer` или `URL`). Если файл с указанным путем назначения существует, то по умолчанию он будет перезаписан. Чтобы в этом случае операция заканчивалась ошибкой, следует в третьем параметре передать флаг `constants.COPYFILE_EXCL`. В параметре `<Режим>` задается комбинация констант `COPYFILE_EXCL`, `COPYFILE_FICLONE` и `COPYFILE_FICLONE_FORCE` через оператор `|`.

(см. описание метода `copyFileSync()`). В последнем параметре указывается функция, которая будет вызвана вне зависимости от успешности операции. Функция принимает один параметр, через который доступен объект ошибки, если скопировать файл не удалось, или значение `null` — в случае успеха.

Пример копирования файла приведен в листинге 18.8.

**Листинг 18.8. Копирование файла функцией `copyFile()`**

```
const fs = require('fs');
const path = require('path');
let srcPath = path.join(__dirname, 'file1.txt');
let destPath = path.join(__dirname, 'folder1', 'file_5.txt');
fs.copyFile(srcPath, destPath, fs.constants.COPYFILE_EXCL,
 function(err) {
 if (err) {
 console.error(err.message);
 return;
 }
 console.log('Файл успешно скопирован');
 });
console.log('Конец программы');
```

Результат в консоли (обратите внимание: инструкции выполняются асинхронно):

```
Конец программы
Файл успешно скопирован
```

- `promises.copyFile()` — асинхронное копирование. Функция возвращает объект `Promise` (см. *разд. 14.1*). Формат функции:

```
<Promise> = promises.copyFile(<Что копируем>, <Куда копируем>[, <Режим>])
```

В первом параметре указывается путь (в виде строки, объекта `Buffer` или URL) к файлу, который требуется скопировать. Если файл не существует, операция заканчивается ошибкой. Во втором параметре указывается путь назначения (в виде строки, объекта `Buffer` или URL). Если файл с указанным путем назначения существует, то по умолчанию он будет перезаписан. Чтобы в этом случае операция заканчивалась ошибкой, следует в третьем параметре передать флаг `constants.COPYFILE_EXCL`. В параметре `<Режим>` перечисляется комбинация констант `COPYFILE_EXCL`, `COPYFILE_FICLONE` и `COPYFILE_FICLONE_FORCE` через оператор `|` (см. описание метода `copyFileSync()`).

Пример копирования файла приведен в листинге 18.9.

**Листинг 18.9. Копирование файла методом `copyFile()`**

```
const fs = require('fs');
const path = require('path');
```



```
(async function() {
 let srcPath = path.join(__dirname, 'file1.txt');
 let destPath = path.join(__dirname, 'folder1', 'file_6.txt');
 try {
 await fs.promises.copyFile(srcPath, destPath,
 fs.constants.COPYFILE_EXCL);
 console.log('Файл успешно скопирован');
 } catch (err) {
 console.error(err.message);
 }
})();
console.log('Конец программы');
```

Результат в консоли (обратите внимание: инструкции выполняются асинхронно):

```
Конец программы
Файл успешно скопирован
```

## 18.3. Удаление файлов

Для удаления файлов предназначены следующие функции из модуля `fs`:

- ❑ `unlinkSync(<Путь к файлу>)` — синхронное удаление. В параметре указывается путь (в виде строки, объекта `Buffer` или `URL`) к удаляемому файлу. При ошибке генерируется исключение. Пример удаления файла приведен в листинге 18.10.

### Листинг 18.10. Удаление файла функцией `unlinkSync()`

```
const fs = require('fs');
const path = require('path');
let p = path.join(__dirname, 'folder1', 'file_6.txt');
try {
 fs.unlinkSync(p);
 console.log('Файл успешно удален');
} catch (err) {
 console.error(err.message);
}
console.log('Конец программы');
```

Результат в консоли (обратите внимание: выполнение инструкций происходит последовательно, т. е. синхронно):

```
Файл успешно удален
Конец программы
```

- ❑ `rmSync(<Путь>[, <Опции>])` — синхронное удаление. В первом параметре указывается путь (в виде строки, объекта `Buffer` или `URL`) к удаляемому файлу. Во втором параметре можно указать объект со следующими свойствами:

- `force` — если `true`, в случае отсутствия файла по указанному пути ничего не происходит, если `false` (значение по умолчанию) — генерируется исключение;
- `maxRetries` — число попыток удаления файла, которые предпримет Node.js, если с первой попытки удалить его не удалось (это может случиться, например, если файл заблокирован какой-либо другой программой). Значение по умолчанию: 0;
- `retryDelay` — задержка между отдельными попытками удаления файла в миллисекундах (значение по умолчанию: 100).

Пример приведен в листинге 18.11.

#### Листинг 18.11. Удаление файла функцией `rmSync()`

```
const fs = require('fs');
const path = require('path');
let p = path.join(__dirname, 'folder1', 'file_6.txt');
try {
 fs.rmSync(p, { force: true });
 console.log('Файл успешно удален');
} catch (err) {
 console.error(err.message);
}
console.log('Конец программы');
```

- `unlink(<Путь к файлу>, <Функция>)` — асинхронное удаление. В первом параметре передается путь (в виде строки, объекта `Buffer` или `URL`) к удаляемому файлу. Во втором параметре указывается функция, которая будет вызвана вне зависимости от успешности операции. Функция принимает один параметр, через который доступен объект ошибки, если удалить файл не удалось, или значение `null` — в случае успеха. Пример приведен в листинге 18.12.

#### Листинг 18.12. Удаление файла функцией `unlink()`

```
const fs = require('fs');
const path = require('path');
let p = path.join(__dirname, 'folder1', 'file_5.txt');
fs.unlink(p, function(err) {
 if (err) {
 console.error(err.message);
 return;
 }
 console.log('Файл успешно удален');
});
console.log('Конец программы');
```

Результат в консоли (обратите внимание: инструкции выполняются асинхронно):

Конец программы

Файл успешно удален

- `rm(<Путь>[, <Опции>], <Функция>)` — асинхронное удаление. Первые два параметра аналогичны таковым у функции `rmSync()`. Третьим параметром указывается функция, которая будет вызвана вне зависимости от успешности операции. Функция принимает один параметр, через который доступен объект ошибки, если удалить файл не удалось, или значение `null` — в случае успеха. Пример приведен в листинге 18.13.

#### Листинг 18.13. Удаление файла функцией `rm()`

```
const fs = require('fs');
const path = require('path');
let p = path.join(__dirname, 'folder1', 'file_5.txt');
fs.rm(p, function(err) {
 if (err) {
 console.error(err.message);
 return;
 }
 console.log('Файл успешно удален');
});
console.log('Конец программы');
```

- `promises.unlink(<Путь к файлу>)` — асинхронное удаление. В параметре передается путь (в виде строки, объекта `Buffer` или `URL`) к удаляемому файлу. Функция возвращает объект `Promise` (см. *разд. 14.1*). Пример удаления файла приведен в листинге 18.14.

#### Листинг 18.14. Удаление файла методом `unlink()`

```
const fs = require('fs');
const path = require('path');
(async function() {
 let p = path.join(__dirname, 'folder1', 'file_4.txt');
 try {
 await fs.promises.unlink(p);
 console.log('Файл успешно удален');
 } catch (err) {
 console.error(err.message);
 }
})();
console.log('Конец программы');
```

Результат в консоли (обратите внимание: инструкции выполняются асинхронно):

Конец программы

Файл успешно удален

- ❑ `promises.rm(<Путь>[, <Опции>])` — асинхронное удаление. Параметры аналогичны таковым у функции `rmSync()`. Функция возвращает объект `Promise` (см. *разд. 14.1*). Пример приведен в листинге 18.15.

#### Листинг 18.15. Удаление файла методом `rm()`

```
const fs = require('fs');
const path = require('path');
(async function() {
 let p = path.join(__dirname, 'folder1', 'file_4.txt');
 try {
 await fs.promises.rm(p);
 console.log('Файл успешно удален');
 } catch (err) {
 console.error(err.message);
 }
})();
console.log('Конец программы');
```

#### ПРИМЕЧАНИЕ

Функции `rmSync()`, `rm()` и `promises.rm()` также можно использовать для удаления каталогов (подробности — далее, в *разд. 18.9*).

## 18.4. Получение сведений о файлах и каталогах

Получить размер файла, время создания, изменения и доступа к файлу, а также другие сведения позволяют следующие функции из модуля `fs`:

- ❑ `statSync(<Путь к файлу>[, <Опции>])` — синхронное получение сведений. В первом параметре указывается путь к файлу в виде строки, объекта `Buffer` или URL. Во втором параметре можно передать объект со следующими свойствами:
- `bigint` — если `true`, числовые значения будут иметь тип `bigint`, если `false` (значение по умолчанию) — `number`;
  - `throwIfNoEntry` — если `true` (значение по умолчанию), при возникновении ошибки в процессе доступа к файлу будет сгенерировано исключение, если `false` — функция вернет значение `undefined`.

Функция возвращает объект `Stats`. Пример получения размера файла приведен в листинге 18.16.

**Листинг 18.16. Получение размера файла функцией `statSync()`**

```
const fs = require('fs');
const path = require('path');
let p = path.join(__dirname, 'file1.txt');
try {
 let stats = fs.statSync(p);
 console.log(stats.size); // 27
} catch (err) {
 console.error(err.message);
}
```

- ❑ `fstatSync(<Дескриптор>[, <Опции>])` — синхронное получение сведений. В первом параметре указывается дескриптор открытого ранее файла. Во втором параметре можно передать объект со свойством `bigint`. Если свойство `bigint` имеет значение `true` (значение по умолчанию: `false`), то числовые значения будут иметь тип `bigint`. Функция возвращает объект `Stats`. При ошибке генерируется исключение. Пример получения размера файла в виде значения типа `bigint` приведен в листинге 18.17.

**Листинг 18.17. Получение размера файла функцией `fstatSync()`**

```
const fs = require('fs');
const path = require('path');
let fd;
try {
 fd = fs.openSync(path.join(__dirname, 'file1.txt'), 'r');
 let stats = fs.fstatSync(fd, { bigint: true });
 console.log(stats.size); // 27n
} catch (err) {
 console.error(err.message);
} finally {
 if (fd !== undefined) fs.closeSync(fd);
}
```

- ❑ `stat(<Путь к файлу>[, <Опции>], <Функция>)` — асинхронное получение сведений. В первом параметре указывается путь к файлу в виде строки, объекта `Buffer` или `URL`. Во втором параметре можно передать объект со свойством `bigint`. Если свойство `bigint` имеет значение `true` (значение по умолчанию: `false`), то числовые значения будут иметь тип `bigint`. В последнем параметре указывается функция, которая будет вызвана вне зависимости от успешности операции. Функция принимает два параметра. Через первый параметр доступен объект ошибки, если возникла ошибка, или значение `null` — если операция выполнена успешно. Через второй параметр доступен объект `Stats`. Пример получения даты создания файла приведен в листинге 18.18.

**Листинг 18.18. Получение даты создания файла функцией `stat()`**

```
const fs = require('fs');
const path = require('path');
let p = path.join(__dirname, 'file1.txt');
fs.stat(p, { bigint: false }, function(err, stats) {
 if (err) {
 console.error(err.message);
 return;
 }
 console.log(stats.ctime); // 2021-05-13T12:56:13.124Z
});
```

- `fstat(<Дескриптор>[, <Опции>], <Функция>)` — асинхронное получение сведений. В первом параметре указывается дескриптор открытого ранее файла. Во втором параметре можно передать объект со свойством `bigint`. Если свойство `bigint` имеет значение `true` (значение по умолчанию: `false`), то числовые значения будут иметь тип `bigint`. В последнем параметре указывается функция, которая будет вызвана вне зависимости от успешности операции. Функция принимает два параметра. Через первый параметр доступен объект ошибки, если возникла ошибка, или значение `null` — если операция выполнена успешно. Через второй параметр доступен объект `Stats`. Пример получения даты создания файла в виде числа миллисекунд, прошедших с начала эпохи, приведен в листинге 18.19.

**Листинг 18.19. Получение даты создания файла функцией `fstat()`**

```
const fs = require('fs');
const path = require('path');
let fd = fs.openSync(path.join(__dirname, 'file1.txt'), 'r');
fs.fstat(fd, { bigint: false }, function(err, stats) {
 if (err) {
 console.error(err.message);
 fs.closeSync(fd);
 return;
 }
 console.log(stats.ctimeMs); // 1620910573123.6843
 fs.closeSync(fd);
});
```

- `promises.stat(<Путь к файлу>[, <Опции>])` — асинхронное получение сведений. В первом параметре указывается путь к файлу в виде строки, объекта `Buffer` или `URL`. Во втором параметре можно передать объект со свойством `bigint`. Если свойство `bigint` имеет значение `true` (значение по умолчанию: `false`), то числовые значения будут иметь тип `bigint`. Функция возвращает объект `Promise` (см. *разд. 14.1*), который при успешном выполнении операции содержит объект

Stats. Пример получения даты создания файла в виде значения типа `bigint` приведен в листинге 18.20.

**Листинг 18.20. Получение даты создания файла методом `stat()`**

```
const fs = require('fs');
const path = require('path');
(async function() {
 try {
 let p = path.join(__dirname, 'file1.txt');
 let stats = await fs.promises.stat(p, { bigint: true });
 console.log(stats.ctimeMs); // 1620910573123n
 } catch (err) {
 console.error(err.message);
 }
})();
```

- `<FileHandle>.stat([<Опции>])` — асинхронное получение сведений. В качестве `<FileHandle>` указывается дескриптор файла, открытого с помощью функции `promises.open()`. В необязательном параметре можно передать объект со свойством `bigint`. Если свойство `bigint` имеет значение `true` (значение по умолчанию: `false`), то числовые значения будут иметь тип `bigint`. Метод возвращает объект `Promise` (см. *разд. 14.1*), который при успешном выполнении операции содержит объект `Stats`.

Выведем значения всех свойств объекта `Stats` (листинг 18.21).

**Листинг 18.21. Получение всех сведений о файле методом `stat()`**

```
const fs = require('fs');
const path = require('path');
(async function() {
 let fh;
 try {
 let p = path.join(__dirname, 'file1.txt');
 fh = await fs.promises.open(p, 'r');
 let stats = await fh.stat({ bigint: false });
 console.log(stats);
 } catch (err) {
 console.error(err.message);
 } finally {
 if (fh !== undefined) await fh.close();
 }
})();
```

**Результат в консоли:**

```
Stats {
 dev: 2862300489,
```

```
mode: 33206,
nlink: 1,
uid: 0,
gid: 0,
rdev: 0,
blksize: 4096,
ino: 3940649673971268,
size: 8,
blocks: 0,
atimeMs: 1620910668750.0417,
mtimeMs: 1620910573123.6843,
ctimeMs: 1620910573123.6843,
birthtimeMs: 1620895247079.7847,
atime: 2021-05-13T12:57:48.750Z,
mtime: 2021-05-13T12:56:13.124Z,
ctime: 2021-05-13T12:56:13.124Z,
birthtime: 2021-05-13T08:40:47.080Z
}
```

Если свойство `bigint` будет иметь значение `true`, результат станет таким:

```
BigIntStats {
 dev: 2862300489n,
 mode: 33206n,
 nlink: 1n,
 uid: 0n,
 gid: 0n,
 rdev: 0n,
 blksize: 4096n,
 ino: 3940649673971268n,
 size: 8n,
 blocks: 0n,
 atimeMs: 1620910668750n,
 mtimeMs: 1620910573123n,
 ctimeMs: 1620910573123n,
 birthtimeMs: 1620895247079n,
 atimeNs: 1620910668750041800n,
 mtimeNs: 1620910573123684400n,
 ctimeNs: 1620910573123684400n,
 birthtimeNs: 1620895247079784600n,
 atime: 2021-05-13T12:57:48.750Z,
 mtime: 2021-05-13T12:56:13.123Z,
 ctime: 2021-05-13T12:56:13.123Z,
 birthtime: 2021-05-13T08:40:47.079Z
}
```

Рассмотрим основные свойства и методы объектов `Stats` и `BigIntStats` (полный список приведен в документации):



- ❑ `isDirectory()` — возвращает значение `true`, если ресурс является каталогом, и `false` — в противном случае:

```
let stats = fs.statSync('file1.txt');
console.log(stats.isDirectory()); // false
stats = fs.statSync(__dirname);
console.log(stats.isDirectory()); // true
```

- ❑ `isFile()` — возвращает значение `true`, если ресурс является файлом, и `false` — в противном случае:

```
let stats = fs.statSync('file1.txt');
console.log(stats.isFile()); // true
stats = fs.statSync(__dirname);
console.log(stats.isFile()); // false
```

- ❑ `size` — размер файла в байтах:

```
let stats = fs.statSync('file1.txt');
console.log(stats.size); // 27
```

- ❑ `mode` — маска типа и прав доступа:

```
let stats = fs.statSync('file1.txt');
console.log(stats.mode); // 33206
console.log(stats.mode.toString(2)); // 1000000110110110
if ((stats.mode & fs.constants.S_IFDIR) !== 0) {
 console.log('Это каталог');
}
else if ((stats.mode & fs.constants.S_IFREG) !== 0) {
 console.log('Это файл');
}
```

Получить информацию о типе позволяют следующие константы:

```
console.log(fs.constants.S_IFMT); // 61440
console.log(fs.constants.S_IFDIR); // 16384
console.log(fs.constants.S_IFREG); // 32768
console.log(fs.constants.S_IFCHR); // 8192
console.log(fs.constants.S_IFLNK); // 40960
```

- ❑ `ctime`, `ctimeMs` и `ctimeNs` — дата последнего изменения статуса файла в виде соответственно объекта `Date`, числа миллисекунд и числа наносекунд, прошедших с начала эпохи:

```
let stats = fs.statSync('file1.txt');
console.log(stats.ctime); // 2021-05-13T12:56:13.124Z
console.log(stats.ctimeMs); // 1620910573123.6843
stats = fs.statSync('file1.txt', { bigint: true });
console.log(stats.ctimeNs); // 1620910573123684400n
```

- ❑ `birthtime`, `birthtimeMs` и `birthtimeNs` — дата создания файла в виде соответственно объекта `Date`, числа миллисекунд и числа наносекунд, прошедших с начала эпохи:

```
let stats = fs.statSync('file1.txt');
console.log(stats.birthtime); // 2021-05-13T08:40:47.080Z
console.log(stats.birthtimeMs); // 1620895247079.7847
stats = fs.statSync('file1.txt', { bigint: true });
console.log(stats.birthtimeNs); // 1620895247079784600n
```

- ❑ `mtime`, `mtimeMs` и `mtimeNs` — дата последнего изменения содержимого файла в виде соответственно объекта `Date`, числа миллисекунд и числа наносекунд, прошедших с начала эпохи:

```
let stats = fs.statSync('file1.txt');
console.log(stats.mtime); // 2021-05-13T12:56:13.124Z
console.log(stats.mtimeMs); // 1620910573123.6843
stats = fs.statSync('file1.txt', { bigint: true });
console.log(stats.mtimeNs); // 1620910573123684400n
```

- ❑ `atime`, `atimeMs` и `atimeNs` — дата последнего доступа к файлу в виде соответственно объекта `Date`, числа миллисекунд и числа наносекунд, прошедших с начала эпохи:

```
let stats = fs.statSync('file1.txt');
console.log(stats.atime); // 2021-05-13T12:57:48.750Z
console.log(stats.atimeMs); // 1620910668750.0417
stats = fs.statSync('file1.txt', { bigint: true });
console.log(stats.atimeNs); // 1620910668750041800n
```

Задать время изменения содержимого и последнего доступа к файлу позволяют следующие функции из модуля `fs`:

```
utimesSync(<Путь к файлу>, <atime>, <mtime>)
utimes(<Путь к файлу>, <atime>, <mtime>, function(err) { })
<Promise> = promises.utimes(<Путь к файлу>, <atime>, <mtime>)
futimesSync(<Дескриптор>, <atime>, <mtime>)
futimes(<Дескриптор>, <atime>, <mtime>, function(err) { })
<Promise> = <FileHandle>.utimes(<atime>, <mtime>)
```

Пример использования функции `utimesSync()` приведен в листинге 18.22.

**Листинг 18.22. Задание времени изменения содержимого и последнего доступа к файлу функцией `utimesSync()`**

```
const fs = require('fs');
try {
 let p = 'file1.txt';
 let stats = fs.statSync(p);
 console.log(stats.atime); // 2021-05-13T12:57:48.750Z
 console.log(stats.mtime); // 2021-05-13T12:56:13.124Z
 fs.utimesSync(p, new Date(), new Date());
 stats = fs.statSync(p);
 console.log(stats.atime); // 2021-05-14T13:03:21.501Z
 console.log(stats.mtime); // 2021-05-14T13:03:21.501Z
}
```

```

} catch (err) {
 console.error(err.message);
}

```

## 18.5. Права доступа к файлу и каталогу

В операционной системе семейства UNIX каждому файлу и каталогу назначаются права доступа отдельно для владельца, группы и прочих. Могут быть назначены отдельные права доступа на чтение, запись и выполнение.

Права доступа обозначаются буквами:

- `r` — файл можно читать, содержимое каталога — просматривать;
- `w` — файл можно модифицировать, удалять и переименовывать, в каталоге можно создавать и удалять файлы, сам каталог можно переименовать или удалить;
- `x` — файл можно запускать на выполнение, в каталоге можно выполнять операции над файлами, в том числе поиск файлов.

Права доступа к файлу кодируются строкой вида:

```
-rw-r--r--
```

Первый символ (`-`) означает, что это файл. Далее три символа (`rw-`) задают права доступа для владельца файла соответственно на чтение (в нашем случае — доступно), запись (доступна) и выполнение (символ `-` означает, что выполнение файла недоступно). Следующие три символа (`r--`) назначают права доступа для группы (у нас — только чтение). Последние три символа (`r--`) задают права для всех остальных пользователей (также только чтение).

Права доступа к каталогу кодируются строкой вида:

```
drwxr-xr-x
```

Первая буква (`d`) означает, что это каталог. Владелец может выполнять в каталоге любые действия (`rwX` — чтение, запись и выполнение), а группа и все остальные пользователи — только читать и производить поиск (`r-x`). Чтобы каталог можно было просматривать, должны быть установлены права на выполнение (`x`).

Кроме того, права доступа могут обозначаться числом, называемым *маской прав доступа* и состоящим из трех цифр от 0 до 7. Первая цифра задает права для владельца, вторая — для группы, а третья — для всех остальных пользователей. Например, права доступа `rw-r--r--` можно записать так: `110 100 100`, что переводится в восьмеричное число `0o644`. Таким образом, если право предоставлено, то в соответствующей позиции стоит 1, а если нет — то 0, например:

```

let n = 0o644;
console.log(n.toString(2)); // 110100100

```

Для проверки прав доступа используются следующие функции из модуля `fs`:

- `accessSync(<Путь>[, <Права>])` — проверяет заданный путь на соответствие указанным правам доступа синхронно. Путь указывается в виде строки, объекта

Buffer или URL. Если права есть, функция вернет значение `undefined`, в противном случае генерируется исключение. Пример проверки возможности записи в файл приведен в листинге 18.23.

**Листинг 18.23. Проверка возможности записи в файл функцией `accessSync()`**

```
const fs = require('fs');
try {
 fs.accessSync('file1.txt', fs.constants.W_OK);
 console.log('Файл доступен для записи');
} catch (err) {
 console.error(err.message);
}
```

- `access(<Путь>[, <Права>], <Функция>)` — проверяет заданный путь на соответствие указанным правам доступа асинхронно. Путь указывается в виде строки, объекта `Buffer` или URL. В последнем параметре задается функция, которая будет вызвана вне зависимости от успешности операции. Функция принимает один параметр, через который доступен объект ошибки, если прав нет, или значение `null` — если права есть. Пример проверки существования файла приведен в листинге 18.24.

**Листинг 18.24. Проверка существования файла функцией `access()`**

```
const fs = require('fs');
fs.access('file1.txt', fs.constants.F_OK, function(err) {
 if (err) console.log('Файл не существует');
 else console.log('Файл существует');
});
```

- `promises.access(<Путь>[, <Права>])` — проверяет заданный путь на соответствие указанным правам доступа асинхронно. Путь указывается в виде строки, объекта `Buffer` или URL. Функция возвращает объект `Promise` (см. *разд. 14.1*). Пример проверки возможности чтения и записи файла приведен в листинге 18.25.

**Листинг 18.25. Проверка возможности чтения и записи файла методом `access()`**

```
const fs = require('fs');
(async function() {
 try {
 await fs.promises.access('file1.txt',
 fs.constants.R_OK | fs.constants.W_OK);
 console.log('Файл доступен для чтения и записи');
 } catch (err) {
 console.error('Нет доступа');
 }
})();
```

**ПРИМЕЧАНИЕ**

Успешная проверка на чтение или запись еще не гарантирует, что файл в действительности можно прочитать или изменить (например, файл может быть заблокирован другой программой).

В параметре <Права> задается одно из следующих значений:

- ☐ 0 — проверка существования файла или каталога (значение по умолчанию);
- ☐ 1 — проверка возможности выполнения (в Windows не используется);
- ☐ 2 — проверка возможности записи в файл;
- ☐ 4 — проверка возможности чтения файла;
- ☐ 6 — проверка возможности чтения и записи файла.

Вместо чисел можно указать следующие константы или их комбинацию через оператор |:

```
console.log(fs.constants.F_OK); // 0
console.log(fs.constants.X_OK); // 1
console.log(fs.constants.W_OK); // 2
console.log(fs.constants.R_OK); // 4
console.log(fs.constants.R_OK | fs.constants.W_OK); // 6
```

Чтобы изменить права доступа, следует воспользоваться следующими функциями из модуля fs:

```
chmodSync(<Путь>, <Права доступа>)
chmod(<Путь>, <Права доступа>, function(err) { })
<Promise> = promises.chmod(<Путь>, <Права доступа>)
fchmodSync(<Дескриптор>, <Права доступа>)
fchmod(<Дескриптор>, <Права доступа>, function(err) { })
<Promise> = <FileHandle>.chmod(<Права доступа>)
```

В параметре <Права доступа> указывается восьмеричное число, являющееся маской прав доступа. В Windows можно изменить только право на чтение. Пример указания у файла права только на чтение и только его владельцем:

```
const fs = require('fs');
try {
 fs.chmodSync('file1.txt', 0o400);
} catch (err) {
 console.error(err.message);
}
```

## 18.6. Проверка существования файлов и каталогов

Для проверки существования файла или каталога, расположенного по заданному пути, используется функция `existsSync(<Путь>)` из модуля `fs`, которая возвращает значение `true`, если путь существует, и `false` — в противном случае. Путь указыва-

ется в виде строки, объекта `Buffer` или `URL`. Операция выполняется синхронно. Пример:

```
const fs = require('fs');
console.log(fs.existsSync('file1.txt')); // true
console.log(fs.existsSync('file8.txt')); // false
```

Также можно воспользоваться функциями `accessSync()`, `access()` и `promises.access()` (см. *разд. 18.5*) со значением `0` (`F_OK`; значение по умолчанию) в параметре `<Права>`:

```
const fs = require('fs');
fs.access('file1.txt', fs.constants.F_OK, function(err) {
 if (err) console.log('Файл не существует');
 else console.log('Файл существует');
});
```

## 18.7. Создание каталогов

Для создания каталога используются следующие функции из модуля `fs`:

□ `mkdirSync(<Путь>[, <Опции>])` — синхронное создание каталога. Путь к создаваемому каталогу указывается в виде строки, объекта `Buffer` или `URL`. В параметре `<Опции>` можно задать маску прав доступа или объект со следующими свойствами:

- `recursive` — если указано значение `true`, будут созданы все каталоги в пути (значение по умолчанию: `false`);
- `mode` — режим доступа к каталогу (значение по умолчанию: `0o777`). В Windows не используется.

Возвращает `undefined`, если свойству `recursive` было дано значение `false`, или строку с путем к первому из созданных каталогов, если было дано значение `true`. Если каталог создать не удалось, генерируется исключение. Пример приведен в листинге 18.26.

### Листинг 18.26. Создание каталога функцией `mkdirSync()`

```
const fs = require('fs');
const path = require('path');
let p = path.join(__dirname, 'folder2', 'folder3');
try {
 const d = fs.mkdirSync(p, { recursive: true });
 console.log('Каталог создан:', d);
 // Каталог создан: C:\book\p2\folder2
} catch (err) {
 console.error(err);
}
```

- ❑ `mkdir(<Путь>[, <Опции>], <Функция>)` — асинхронное создание каталога. Первые два параметра аналогичны таковым у функции `mkdirSync()`. В последнем параметре указывается функция, которая будет вызвана вне зависимости от успешности операции. Функция принимает два параметра. Через первый параметр доступен объект ошибки, если каталог создать не удалось, или значение `null` — в случае успеха. Через второй параметр доступно значение `undefined`, если свойству `recursive` было дано значение `false`, или строка с путем к первому из созданных каталогов, если это свойство получило значение `true`. Пример приведен в листинге 18.27.

**Листинг 18.27. Создание каталога функцией `mkdir()`**

```
const fs = require('fs');
const path = require('path');
let p = path.join(__dirname, 'folder4', 'folder5');
fs.mkdir(p, { recursive: true }, function(err, d) {
 if (err) throw err;
 console.log('Каталог создан:', d);
 // Каталог создан: C:\book\p2\folder4
});
```

- ❑ `promises.mkdir(<Путь>[, <Опции>])` — асинхронное создание каталога. Параметры аналогичны таковым у функции `mkdirSync()`. Функция возвращает объект `Promise` (см. *разд. 14.1*), содержащий значение `undefined`, если свойству `recursive` было дано значение `false`, или строку с путем к первому из созданных каталогов, если это свойство получило значение `true`. Пример приведен в листинге 18.28.

**Листинг 18.28. Создание каталога методом `mkdir()`**

```
const fs = require('fs');
const path = require('path');
let p = path.join(__dirname, 'folder6', 'folder7');
(async function() {
 try {
 const d = await fs.promises.mkdir(p, { recursive: true });
 console.log('Каталог создан:', d);
 // Каталог создан: C:\book\p2\folder6
 } catch (err) {
 console.error(err);
 }
})();
```

**ПРИМЕЧАНИЕ**

Для переименования каталогов следует применять функции `renameSync()`, `rename()` и `promises.rename()`. Описание функций приведено в *разд. 18.1*.

## 18.8. Создание временных каталогов

Получить путь к каталогу для временных файлов позволяет функция `tmpdir()` из модуля `os`:

```
const os = require('os');
console.log(os.tmpdir());
// C:\Users\<Имя пользователя>\AppData\Local\Temp
```

Для создания временного каталога предназначены следующие функции из модуля `fs`:

- ❑ `mkdtempSync(<Префикс>[, <Опции>])` — синхронное создание временного каталога. В первом параметре указывается префикс, к которому будут добавлены случайные символы для создания уникального имени каталога. Если нужно создать имя без префикса, значение параметра `<Префикс>` должно завершаться символом, используемым в системе для разделения каталогов в пути (`path.sep`). В параметре `<Опции>` можно указать кодировку в виде строки (значение по умолчанию: `'utf8'`) или объекта со свойством `encoding`. При успешном выполнении функция возвращает путь к созданному каталогу, при неудаче генерируется исключение. Пример приведен в листинге 18.29.

### Листинг 18.29. Создание временного каталога функцией `mkdtempSync()`

```
const os = require('os');
const fs = require('fs');
const path = require('path');
let p = path.join(os.tmpdir(), 'test-');
try {
 let name = fs.mkdtempSync(p, { encoding: 'utf8' });
 console.log(name);
} catch (err) {
 console.error(err);
}
```

Результат:

C:\Users\<Имя пользователя>\AppData\Local\Temp\test-UWIK7J

- ❑ `mkdtemp(<Префикс>[, <Опции>], <Функция>)` — асинхронное создание временного каталога. Первые два параметра аналогичны таковым у функции `mkdtempSync()`. В последнем параметре указывается функция, которая будет вызвана вне зависимости от успешности операции. Функция принимает два параметра. Через первый параметр доступен объект ошибки, если каталог создать не удалось, или значение `null` — если каталог успешно создан. Через второй параметр доступен путь к созданному каталогу в случае успешного выполнения. Пример приведен в листинге 18.30.



**Листинг 18.30. Создание временного каталога функцией `mkdtemp()`**

```
const os = require('os');
const fs = require('fs');
const path = require('path');
fs.mkdtemp(os.tmpdir() + path.sep, function(err, name) {
 if (err) throw err;
 console.log(name);
});
```

**Результат:**

C:\Users\<Имя пользователя>\AppData\Local\Temp\ZhDmgb

- `promises.mkdtemp(<Префикс>[, <Опции>])` — асинхронное создание временного каталога. Параметры аналогичны таковым у функции `mkdtempSync()`. Функция возвращает объект `Promise` (см. *разд. 14.1*), содержащий путь к созданному каталогу. Пример приведен в листинге 18.31.

**Листинг 18.31. Создание временного каталога методом `mkdtemp()`**

```
const fs = require('fs');
const path = require('path');
(async function() {
 try {
 let name = await fs.promises.mkdtemp(__dirname + path.sep);
 console.log(name);
 } catch (err) {
 console.error(err);
 }
})();
```

**Результат:**

C:\book\p1\heDyrp

## 18.9. Удаление каталогов

Для удаления каталога применяются следующие функции из модуля `fs`:

- `rmdirSync(<Путь>[, <Опции>])` — синхронное удаление каталога. Путь к удаляемому каталогу указывается в виде строки, объекта `Buffer` или URL. В параметре `<Опции>` можно передать объект со следующими свойствами:
- `maxRetries` — число попыток удаления каталога, которые предпримет Node.js, если с первой попытки удалить его не удалось (это может случиться, например, если каталог или файлы в нем заблокированы какой-либо другой программой). Значение по умолчанию: 0;

- `retryDelay` — задержка между отдельными попытками удаления каталога в миллисекундах (значение по умолчанию: 100).

Если каталог удалить не удалось или выполняется попытка удалить файл, генерируется исключение.

#### ОБРАТИТЕ ВНИМАНИЕ

Удалить можно только пустой каталог.

Пример приведен в листинге 18.32.

#### Листинг 18.32. Удаление каталога функцией `rmdirSync()`

```
const fs = require('fs');
const path = require('path');
try {
 fs.rmdirSync(path.join(__dirname, 'folder4'));
 console.log('Каталог удален');
} catch (err) {
 console.error(err);
}
```

Для удаления каталогов, в том числе и непустых (содержащих другие каталоги и файлы), можно использовать функцию `rmSync()` (см. *разд. 18.3*). Для этого необходимо указать первым параметром путь к удаляемому каталогу, а в объект, задаваемый во втором параметре, добавить свойство `recursive` со значением `true` (листинг 18.33).

#### Листинг 18.33. Удаление каталога функцией `rmSync()`

```
const fs = require('fs');
const path = require('path');
try {
 fs.rmSync(path.join(__dirname, 'folder4'),
 { recursive: true });
 console.log('Каталог удален');
} catch (err) {
 console.error(err);
}
```

- `rm(<Путь>[, <Опции>], <Функция>)` — асинхронное удаление каталога, в том числе непустого. Функция была описана в *разд. 18.3*. Первым параметром указывается путь к удаляемому каталогу, а в объект, задаваемый во втором параметре, следует добавить свойство `recursive` со значением `true` (листинг 18.34).

#### Листинг 18.34. Удаление каталога функцией `rm()`

```
const fs = require('fs');
const path = require('path');
```

```
fs.rm(path.join(__dirname, 'folder6'),
 { recursive: true }, function(err) {
 if (err) throw err;
 console.log('Каталог удален');
 });
```

- ❑ `promises.rmdir(<Путь>[, <Опции>])` — асинхронное удаление каталога. Параметры аналогичны таковым у функции `rmdirSync()`. Функция возвращает объект `Promise` (см. *разд. 14.1*). Пример приведен в листинге 18.35.

#### Листинг 18.35. Удаление каталога методом `rmdir()`

```
const fs = require('fs');
const path = require('path');
let p = path.join(__dirname, 'folder2', 'folder3');
(async function() {
 try {
 await fs.promises.rmdir(p);
 console.log('Каталог удален');
 } catch (err) {
 console.error(err);
 }
})();
```

В этом примере будет удален только каталог `folder3`. При попытке удалить каталог `folder2` будет сгенерировано исключение, т. к. каталог не является пустым. Помните: удалить можно только пустой каталог.

Для удаления каталогов, в том числе и непустых, можно использовать функцию `promises.rm()` (см. *разд. 18.3*). Для этого необходимо указать первым параметром путь к удаляемому каталогу, а в объект, задаваемый во втором параметре, добавить свойство `recursive` со значением `true`. Пример приведен в листинге 18.36.

#### Листинг 18.36. Удаление каталога методом `rm()`

```
const fs = require('fs');
const path = require('path');
let p = path.join(__dirname, 'folder2', 'folder3');
(async function() {
 try {
 await fs.promises.rm(p, { recursive: true });
 console.log('Каталог удален');
 } catch (err) {
 console.error(err);
 }
})();
```

## 18.10. Перебор элементов в каталоге

Чтобы перебрать элементы (файлы и каталоги), расположенные в каком-либо каталоге, сначала следует открыть этот каталог на чтение. Для этого применяются следующие функции из модуля `fs`:

- `opendirSync(<Путь>[, <Опции>])` — синхронное открытие каталога. Путь к открываемому каталогу указывается в виде строки, объекта `Buffer` или URL. В параметре `<Опции>` можно передать объект со следующими свойствами:
  - `bufferSize` — число прочитанных из каталога элементов, сохраняемых во внутреннем буфере. Значение по умолчанию: 32. Увеличение этого параметра приводит к повышению производительности, но повышает объем занимаемой оперативной памяти;
  - `encoding` — кодировка, в которой будут представлены имена прочитанных из каталога элементов (по умолчанию: `'utf8'`). Если указано значение `'buffer'`, имена элементов будут храниться в виде объектов `Buffer`.

Если каталог открыть не удалось, генерируется исключение. В противном случае возвращается объект `Dir`, позволяющий перебрать элементы каталога. Пример приведен в листинге 18.37.

### Листинг 18.37. Перебор элементов каталога функцией `opendirSync()`

```
const fs = require('fs');
const path = require('path');
try {
 let dir = fs.opendirSync(path.join(__dirname, 'folder1'));
 console.log(dir.path); // C:\book\p1\folder1
 console.log(dir.readSync());
 // Dirent { name: 'file_1.txt', [Symbol(type)]: 1 }
 console.log(dir.readSync());
 // Dirent { name: 'file_2.txt', [Symbol(type)]: 1 }
 console.log(dir.readSync());
 // Dirent { name: 'file_3.txt', [Symbol(type)]: 1 }
 console.log(dir.readSync()); // null
} catch (err) {
 console.error(err);
}
```

- `opendir(<Путь>[, <Опции>], <Функция>)` — асинхронное открытие каталога. Первые два параметра аналогичны таковым у функции `opendirSync()`. В последнем параметре указывается функция, которая будет вызвана вне зависимости от успешности операции. Функция принимает два параметра. Через первый параметр доступен объект ошибки, если каталог открыть не удалось, или значение `null` — если каталог успешно открыт. Через второй параметр доступен объект `Dir`. Пример приведен в листинге 18.38.

**Листинг 18.38. Перебор элементов каталога функцией `opendir()`**

```
const fs = require('fs');
const path = require('path');
let p = path.join(__dirname, 'folder1');
fs.opendir(p, function(err, dir) {
 if (err) throw err;
 let d;
 while ((d = dir.readSync())) {
 console.log(d.name);
 }
});
```

- ❑ `promises.opendir(<Путь>[, <Опции>])` — асинхронное открытие каталога. Параметры аналогичны таковым у функции `opendirSync()`. Функция возвращает объект `Promise` (см. *разд. 14.1*), который содержит объект `Dir`. Пример приведен в листинге 18.39.

**Листинг 18.39. Перебор элементов каталога методом `opendir()`**

```
const fs = require('fs');
const path = require('path');
let p = path.join(__dirname, 'folder1');
(async function() {
 try {
 let dir = await fs.promises.opendir(p);
 for await (const d of dir) {
 console.log(d.name);
 }
 } catch (err) {
 console.error(err);
 }
})();
```

Открытый на чтение каталог представляется объектом класса `Dir`, который содержит следующие основные свойства и методы:

- ❑ `path` — путь к каталогу;
- ❑ `readSync()` — синхронное чтение очередного элемента каталога. Возвращает прочитанный элемент в виде объекта класса `Dirent` или значение `null`, если элементов больше нет. Пример использования приведен в описании к функции `opendirSync()`;
- ❑ `read(<Функция>)` — асинхронное чтение очередного элемента каталога. Вызывает заданную функцию, передавая ей два параметра: через первый доступен объект ошибки или значение `null`, через второй — прочитанный элемент в виде объекта класса `Dirent` или значение `null`, если элементов больше нет. Пример приведен в листинге 18.40.

**Листинг 18.40. Чтение элементов каталога методом read() 1**

```
const fs = require('fs');
const path = require('path');
let p = path.join(__dirname, 'folder1');
fs.opendir(p, function(err, dir) {
 if (err) throw err;
 dir.read(function(e, d) {
 if (e) throw e;
 console.log(d.name); // file_1.txt
 });
});
```

- ❑ read() — асинхронное чтение очередного элемента каталога. Возвращает объект Promise (см. *разд. 14.1*), который содержит прочитанный элемент в виде объекта Dirent или значение null, если элементов больше нет. Пример приведен в листинге 18.41.

**Листинг 18.41. Чтение элементов каталога методом read() 2**

```
const fs = require('fs');
const path = require('path');
let p = path.join(__dirname, 'folder1');
(async function() {
 try {
 let dir = await fs.promises.opendir(p);
 let d = await dir.read();
 if (d) console.log(d.name); // file_1.txt
 await dir.close();
 } catch (err) {
 console.error(err);
 }
})();
```

- ❑ [Symbol.asyncIterator]() — возвращает асинхронный итератор, с помощью которого можно перебрать все элемента каталога (листинг 18.42).

**Листинг 18.42. Чтение элементов каталога с помощью итератора**

```
const fs = require('fs');
const path = require('path');
let p = path.join(__dirname, 'folder1');
(async function() {
 try {
 let dir = await fs.promises.opendir(p);
 let it = dir[Symbol.asyncIterator]();
 console.log(await it.next());
 }
})();
```

```

 // {
 // value: Dirent { name: 'file_1.txt',
 // [Symbol(type)]: 1 },
 // done: false
 // }
 for await (const d of dir) {
 console.log(d.name);
 }
 } catch (err) {
 console.error(err);
 }
}());

```

- ❑ `closeSync()` — закрывает каталог синхронно;
- ❑ `close(<Функция>)` — закрывает каталог асинхронно. Заданная функция через единственный параметр принимает объект ошибки или значение `null`;
- ❑ `close()` — закрывает каталог асинхронно. Возвращает объект `Promise` (см. *разд. 14.1*).

Объект класса `Dirent`, представляющий элемент каталога, содержит следующие основные свойства и методы (полный список приведен в документации):

- ❑ `name` — имя элемента каталога в виде строки или объекта `Buffer` (зависит от значения свойства `encoding`, указываемой в объекте `<Опции>` при вызове функции `opendirSync()`, `opendir()` или методом `promises.opendir()`);
- ❑ `isDirectory()` — возвращает значение `true`, если элемент является каталогом, и `false` — в противном случае;
- ❑ `isFile()` — возвращает значение `true`, если элемент является файлом, и `false` — в противном случае (листинг 18.43).

#### Листинг 18.43. Использование методов `isDirectory()` и `isFile()`

```

const fs = require('fs');
(async function() {
 try {
 let dir = await fs.promises.opendir(__dirname);
 for await (const d of dir) {
 console.log(d.name);
 console.log('dir:', d.isDirectory());
 console.log('file:', d.isFile());
 }
 } catch (err) {
 console.error(err);
 }
}());

```

Получить массив со всеми элементами внутри каталога позволяют следующие функции из модуля `fs`:

□ `readdirSync(<Путь>[, <Опции>])` — синхронное получение элементов каталога. Путь указывается в виде строки, объекта `Buffer` или URL. В параметре `<Опции>` можно указать кодировку или передать объект со следующими свойствами:

- `encoding` — кодировка, в которой будут представлены имена прочитанных из каталога элементов (значение по умолчанию: `'utf8'`). Если указано значение `'buffer'`, имена элементов будут представляться объектами `Buffer`;
- `withFileTypes` — если указано значение `true`, возвращаемый функцией массив будет состоять из объектов `Dirent`, если `false` (значение по умолчанию) — из обычных строк.

Результат возвращается в виде массива с элементами каталога. Пример приведен в листинге 18.44.

**Листинг 18.44. Получение всех элементов каталога функцией `readdirSync()`**

```
const fs = require('fs');
const path = require('path');
let p = path.join(__dirname, 'folder1');
let arr = fs.readdirSync(p, { encoding: 'utf8', withFileTypes: false });
console.log(arr);
// ['file_1.txt', 'file_2.txt', 'file_3.txt']
```

□ `readdir(<Путь>[, <Опции>], <Функция>)` — асинхронное получение элементов каталога. Первые два параметра аналогичны таковым у функции `readdirSync()`. В последнем параметре указывается функция, которая будет вызвана вне зависимости от успешности операции. Функция принимает два параметра: через первый доступен объект ошибки или значение `null`, через второй — массив с элементами. Пример приведен в листинге 18.45.

**Листинг 18.45. Получение всех элементов каталога функцией `readdir()`**

```
const fs = require('fs');
const path = require('path');
let p = path.join(__dirname, 'folder1');
fs.readdir(p, 'buffer', function(err, arr) {
 if (err) throw err;
 console.log(arr);
 // [
 // <Buffer 66 69 6c 65 5f 31 2e 74 78 7a>,
 // <Buffer 66 69 6c 65 5f 32 2e 74 78 7a>,
 // <Buffer 66 69 6c 65 5f 33 2e 74 78 7a>
 //]
});
```



- ❑ `promises.readdir(<Путь>[, <Опции>])` — асинхронное получение элементов каталога. Параметры аналогичны таковым у функции `readdirSync()`. Функция возвращает объект `Promise` (см. *разд. 14.1*), который содержит массив с элементами. Пример приведен в листинге 18.46.

**Листинг 18.46. Получение всех элементов каталога методом `readdir()`**

```
const fs = require('fs');
const path = require('path');
let p = path.join(__dirname, 'folder1');
(async function() {
 try {
 let arr = await fs.promises.readdir(p, { withFileTypes: true });
 console.log(arr);
 // [
 // Dirent { name: 'file_1.txt', [Symbol(type)]: 1 },
 // Dirent { name: 'file_2.txt', [Symbol(type)]: 1 },
 // Dirent { name: 'file_3.txt', [Symbol(type)]: 1 }
 //]
 } catch (err) {
 console.error(err);
 }
})();
```

## 18.11. Отслеживание изменения файла или каталога

Отследить изменение файла или каталога позволяют следующие функции из модуля `fs`:

- ❑ `watch(<Путь>[, <Опции>][, <Обработчик>])` — запускает отслеживание изменений файла или каталога, расположенного по заданному пути, который должен быть строкой, объектом `Buffer` или URL. Во втором параметре можно передать кодировку в виде строки или объект со следующими свойствами:
- `encoding` — кодировка имени файла (каталога) в виде строки (значение по умолчанию: `'utf8'`);
  - `persistent` — если указано значение `true`, то процесс будет продолжать работать, пока изменения отслеживаются (значение по умолчанию: `true`);
  - `recursive` — если указано значение `true`, то будет отслеживаться содержимое вложенных каталогов, если задан путь к каталогу (значение по умолчанию: `false`).

В последнем параметре указывается функция, которая будет вызвана после изменения файла (каталога). Функция принимает два параметра: через первый

доступен тип события в виде строки, через второй — имя файла (каталога) в виде строки или объекта `Buffer`. Функция `watch()` возвращает объект `FSWatcher`.

Пример отслеживания изменения файла приведен в листинге 18.47 (для остановки программы нажмите комбинацию клавиш `<Ctrl>+<C>`).

**Листинг 18.47. Отслеживание изменений файла функцией `watch()`**

```
const fs = require('fs');
const path = require('path');
let p = path.join(__dirname, 'file1.txt');
let obj = fs.watch(p, {
 encoding: 'utf8', recursive: false, persistent: true
}, (e, fileName) => {
 console.log(`event: ${e}`);
 console.log(`fileName: ${fileName}`);
});
console.log(obj);
```

**Результат в консоли после изменения содержимого файла:**

```
C:\book\p1>node index.js
<ref *1> FSWatcher {
 _events: [Object: null prototype] {
 { change: [Function (anonymous)] },
 _eventsCount: 1,
 _maxListeners: undefined,
 _handle: FSEvent {
 onchange: [Function (anonymous)],
 [Symbol(owner_symbol)]: [Circular *1]
 },
 [Symbol(kCapture)]: false
 },
 event: change
 fileName: file1.txt
 event: change
 fileName: file1.txt
```

Объект класса `FSWatcher` содержит метод `close()`, который останавливает отслеживание и поддерживает следующие события:

- `close` — генерируется после остановки отслеживания изменений в заданном файле или каталоге (т. е. после вызова метода `close()` объекта);
- `change` — генерируется при каждом изменении заданного файла или каталога. В функцию-обработчик передаются два параметра: через первый доступен тип события в виде строки, через второй — имя файла (каталога) в виде строки или объекта `Buffer`;

- `error` — генерируется при возникновении ошибки. В функцию-обработчик передается объект ошибки.

Обработаем события и остановим отслеживание через 5 секунд (листинг 18.48).

#### Листинг 18.48. Останов отслеживания файла

```
const fs = require('fs');
const path = require('path');
let p = path.join(__dirname, 'file1.txt');
let watcher = fs.watch(p);
console.log('Отслеживание запущено');
setTimeout(function() {
 watcher.close();
}, 5000);
watcher.on('close', () => {
 console.log('Обработчик удален');
});
watcher.on('change', (e, fileName) => {
 console.log(`event: ${e}`);
 console.log(`fileName: ${fileName}`);
});
```

□ `watchFile(<Путь>[, <Опции>], <Обработчик>)` — запускает отслеживание изменения файла, расположенного по указанному пути, который должен быть строкой, объектом `Buffer` или `URL`. Во втором параметре можно передать объект со следующими свойствами:

- `interval` — интервал опроса в миллисекундах (значение по умолчанию: 5007);
- `persistent` — если указано значение `true`, то процесс будет продолжать работать, пока изменения отслеживаются (значение по умолчанию: `true`);
- `bigint` — если указано значение `true`, то числовые значения объекта `Stat` будут иметь тип `bigint` (значение по умолчанию: `false`).

В последнем параметре указывается функция, которая будет вызвана после изменения файла. Функция принимает два параметра: через первый доступен текущий объект `Stat` (см. *разд. 18.4*), а через второй — предыдущий объект `Stat`. Функция `watchFile()` возвращает объект `StatWatcher`.

Пример отслеживания изменения файла приведен в листинге 18.49 (для остановки программы нажмите комбинацию клавиш `<Ctrl>+<C>`).

#### Листинг 18.49. Отслеживание изменений файла функцией `watchFile()`

```
const fs = require('fs');
const path = require('path');
```

```
let p = path.join(__dirname, 'file1.txt');
let obj = fs.watchFile(p, {
 persistent: true, interval: 5007, bigint: false
}, (curStat, prevStat) => {
 console.log(`curStat: ${curStat}`);
 console.log(`prevStat: ${prevStat}`);
});
console.log(obj);
```

**Результат в консоли после изменения содержимого файла:**

```
C:\book\p1>node index.js
<ref *1> StatWatcher {
 _events: [Object: null prototype] ↗
 { change: [Function (anonymous)] },
 _eventsCount: 1,
 _maxListeners: undefined,
 _handle: StatWatcher {
 onchange: [Function: onchange],
 [Symbol(owner_symbol)]: [Circular *1]
 },
 [Symbol(kCapture)]: false,
 [Symbol(kOldStatus)]: -1,
 [Symbol(kUseBigint)]: false,
 [Symbol(KFSStatWatcherRefCount)]: 1,
 [Symbol(KFSStatWatcherMaxRefCount)]: 1
}
curStat: [object Object]
prevStat: [object Object]
```

- ❑ `unwatchFile(<Путь>[, <Обработчик>])` — останавливает отслеживание изменения файла по указанному пути, ранее назначенное функцией `watchFile()`. Путь задается в виде строки, объекта `Buffer` или URL. Во втором параметре указывается ссылка на обработчик, назначенный функцией `watchFile()`. Если второй параметр не указан, то удаляются все обработчики. Пример удаления всех обработчиков после изменения файла:

```
const fs = require('fs');
const path = require('path');
let p = path.join(__dirname, 'file1.txt');
fs.watchFile(p, (curStat, prevStat) => {
 fs.unwatchFile(p);
 console.log('Обработчик удален');
});
```

### **ПРИМЕЧАНИЕ**

Вместо функций `watchFile()` и `unwatchFile()` лучше использовать функцию `watch()`.



## ГЛАВА 19

# Веб-сервер на Node.js

На Node.js можно написать не только консольное приложение, но и веб-сервер, отслеживающий поступление клиентских запросов на какой-либо TCP-порт и отправляющий серверные ответы, основываясь на строке запроса.

### 19.1. Создание и запуск веб-сервера

Создать веб-сервер позволяет функция `createServer()` из модуля `http`. Формат функции:

```
createServer([<Опции>,]<Функция>)
```

В первом необязательном параметре можно передать объект со следующими свойствами (полное описание свойств приведено в документации):

- ☐ `IncomingMessage` — ссылка на класс, используемый для представления входящего запроса (значение по умолчанию: `http.IncomingMessage`);
- ☐ `ServerResponse` — ссылка на класс, применяемый для представления выдаваемого сервером ответа (значение по умолчанию: `http.ServerResponse`);
- ☐ `insecureHTTPParser` — если `true`, будет задействован обработчик входящих запросов, позволяющий обрабатывать некорректно сформированные запросы (вследствие чего сервер может оказаться подверженным сетевым атакам). Если `false` (значение по умолчанию), будет задействован обычный обработчик, требующий корректного формирования запросов;
- ☐ `maxHeaderSize` — максимальный размер заголовков, которые могут присутствовать во входящем запросе (значение по умолчанию: 16 384 байт, или 16 Кбайт).

Как правило, значения этих параметров изменяют в специфических случаях. Мы этого делать не будем.

В параметре `<Функция>` указывается функция обратного вызова, которая вызывается при поступлении очередного клиентского запроса. Через первый параметр досту-

пен объект полученного запроса `http.IncomingMessage`, а через второй — объект отправляемого серверного ответа `http.ServerResponse`. Функция возвращает объект `http.Server`.

Чтобы запустить веб-сервер, нужно вызвать у представляющего его объекта метод `listen()`, передав ему в качестве параметра номер TCP-порта, на котором веб-сервер будет ожидать запросы. Напишем сервер, который будет выводить в консоль строку запроса и отправлять серверный ответ в виде строки, и запустим его на порту 3000 (листинг 19.1).

#### Листинг 19.1. Простейший веб-сервер

```
const http = require('http');
let server = http.createServer(function(req, res) {
 console.log(req.url);
 res.setHeader('Content-Type', 'text/html; charset=utf-8');
 res.write('<h1>Ответ сервера</h1>');
 res.end(req.url);
});
server.listen(3000, () => {
 console.log('Сервер запущен на порту 3000');
});
```

Переходим в командную строку и выполняем команду:

```
C:\book\pl>node index.js
```

Успешно запущенный веб-сервер выведет следующее сообщение:

```
Сервер запущен на порту 3000
```

Запустим веб-браузер и выполним переход по интернет-адресу **`http://localhost:3000/test`**. Браузер отобразит следующую информацию:

```
Ответ сервера
/test
```

В консоли будут выведены следующие строки запросов:

```
/test
/favicon.ico
```

Здесь `test` — это запрашиваемый ресурс, а `favicon.ico` — значок, отображаемый веб-браузером в заголовке вкладки и в **Избранном**. Иногда также запрашивается файл `robots.txt`.

После каждого изменения исходного кода нужно остановить веб-сервер и запустить его снова. Для остановки веб-сервера достаточно в консоли нажать комбинацию клавиш `<Ctrl>+<C>`.

## 19.2. Объект запроса

Через первый параметр функции обратного вызова, передаваемой функции `createServer()`, доступен объект запроса, реализуемый классом `IncomingMessage`. Перечислим основные свойства этого объекта (полный список приведен в документации):

- `url` — строка запроса, идентифицирующая запрашиваемый ресурс (например,  `'/'` или  `'/test'`);
- `method` — HTTP-метод, посредством которого был выполнен запрос (например,  `'GET'`);
- `headers` — объект с заголовками запроса, например:

```
{
 host: 'localhost:3000',
 'user-agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:88.0) Gecko/20100101 Firefox/88.0',
 accept: 'image/webp,*/*',
 'accept-language': 'ru-RU,ru;q=0.8,en-US;q=0.5,en;q=0.3',
 'accept-encoding': 'gzip, deflate',
 connection: 'keep-alive',
 referer: 'http://localhost:3000/test'
}
```

- `rawHeaders` — массив с необработанными заголовками запроса в том порядке, в котором они были получены, например:

```
[
 'Host',
 'localhost:3000',
 'User-Agent',
 'Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:88.0) Gecko/20100101 Firefox/88.0',
 'Accept',
 'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8',
 'Accept-Language',
 'ru-RU,ru;q=0.8,en-US;q=0.5,en;q=0.3',
 'Accept-Encoding',
 'gzip, deflate',
 'Connection',
 'keep-alive',
 'Upgrade-Insecure-Requests',
 '1',
 'Cache-Control',
 'max-age=0'
]
```

- ❑ `httpVersion` — версия протокола HTTP (например, '1.1');
- ❑ `httpVersionMajor` — первое значение в версии протокола;
- ❑ `httpVersionMinor` — второе значение в версии протокола.

Обрабатываем запросы сразу нескольких ресурсов, а в ответ на прочие запросы отправим коды статуса 404 (ресурс не найден), что иллюстрирует листинг 19.2.

#### Листинг 19.2. Веб-сервер, обрабатывающий запросы нескольких ресурсов

```
const http = require('http');
let server = http.createServer(function(req, res) {
 if (req.url === '/' || req.url === '/test') {
 res.setHeader('Content-Type', 'text/html; charset=utf-8');
 res.write('<h1>/ или /test</h1>');
 res.end();
 }
 else if(req.url === '/about') {
 res.setHeader('Content-Type', 'text/html; charset=utf-8');
 res.write('<h1>/about</h1>');
 res.end();
 }
 else {
 res.statusCode = 404;
 res.end('<h1>Not found</h1>');
 }
});
server.listen(3000, () => {
 console.log('Сервер запущен на порту 3000');
});
```

При запросе **http://localhost:3000/** или **http://localhost:3000/test** получим сообщение **"/"** или **"/test"**, а при запросе **http://localhost:3000/about** — сообщение **"/about"**. Код статуса в обоих случаях — 200. При запросе любых других ресурсов получим сообщение **"Not found"** и код статуса 404.

#### ПРИМЕЧАНИЕ

Для более удобной обработки запросов следует использовать сторонние пакеты, например Express.

## 19.3. Объект ответа

Через второй параметр функции обратного вызова доступен объект серверного ответа, реализуемый классом `ServerResponse`. Перечислим основные свойства и методы этого объекта (полный список приведен в документации):

- ❑ `write(<Данные>[, <Кодировка>][, <Функция>])` — записывает указанные данные в поток вывода, который перенесет их в отправляемый серверный ответ. Данные



могут быть в виде строки или объекта `Buffer`. Во втором параметре можно указать кодировку данных (значение по умолчанию: `'utf8'`). В третьем параметре можно задать функцию, которая будет вызвана после успешной записи данных в поток. Возвращает `true`, если данные были успешно записаны в поток, и `false`, если записать их сразу не удалось и данные были временно сохранены во внутреннем буфере и будут записаны позже;

- ❑ `end(<Данные>[, <Кодировка>][, <Функция>])` — записывает данные в поток вывода и завершает работу с ответом. Параметры аналогичны таковым у метода `write()`.

Вызов метода `write()` после вызова метода `end()` приведет к ошибке;

- ❑ `statusCode` — целочисленный код статуса:  
`res.statusCode = 404;`
- ❑ `statusMessage` — текстовое описание статуса:  
`res.statusMessage = 'Запрашиваемый файл не найден';`
- ❑ `writableEnded` — `true`, если был вызван метод `end()` и работа с ответом была завершена, и `false` — в противном случае;
- ❑ `setHeader(<Имя>, <Значение>)` — добавляет в ответ заголовок с заданными именем и значением:  
`res.setHeader('Content-Type', 'text/html; charset=utf-8');`

Во втором параметре можно передать строку или массив строк, если нужно отправить сразу несколько значений, например несколько `cookies`;

- ❑ `getHeaders()` — возвращает объект, свойства которого соответствуют заголовкам ответа и хранят значения этих заголовков;
- ❑ `getHeaderNames()` — возвращает массив с именами заголовков ответа;
- ❑ `hasHeader(<Имя>)` — возвращает `true`, если в ответе присутствует заголовок с заданным именем, и `false` — в противном случае;
- ❑ `getHeader(<Имя>)` — возвращает значение заголовка ответа с указанным именем:  
`console.log(res.getHeader('Content-Type'));`  
`// text/html; charset=utf-8`
- ❑ `writeHead()` — устанавливает код статуса и заносит в ответ заголовки сразу, без промежуточного сохранения в буфере. Формат метода:  
`writeHead(<Код статуса>[, <Описание статуса>], <Заголовки>)`

Код статуса указывается в виде целого числа, необязательное текстовое описание — в виде строки. Заголовки указываются в виде объекта, каждое свойство которого представляет собой один из отправляемых заголовков. Пример:

```
res.writeHead(200, {
 'Content-Type': 'text/html; charset=utf-8'
});
```

Заголовки также можно указать в виде массива, в котором четные элементы хранят имена заголовков, а нечетные — их значения (как и в массиве, возвращаемом методом `rawHeaders()` объекта запроса, описанным в *разд. 19.2*):

```
res.writeHead(200, [
 'Content-Type', 'text/html; charset=utf-8',
 'Platform', 'Node.js'
]);
```

Выполним переадресацию при запросе главной страницы на `/test` (листинг 19.3).

#### Листинг 19.3. Веб-сервер с поддержкой переадресации

```
const http = require('http');
let server = http.createServer(function(req, res) {
 if (req.url === '/') {
 res.statusCode = 302;
 res.setHeader('Location', '/test');
 res.end();
 }
 else if (req.url === '/test') {
 res.setHeader('Content-Type', 'text/html; charset=utf-8');
 res.write('<h1>/test</h1>');
 res.end();
 }
 else {
 res.statusCode = 404;
 res.end('<h1>Not found</h1>');
 }
});
server.listen(3000, () => {
 console.log('Сервер запущен на порту 3000');
});
```

## 19.4. Отправка файла в качестве серверного ответа

Отправить содержимое файла в качестве серверного ответа можно так, как показано в листинге 19.4.

#### Листинг 19.4. Веб-сервер, отправляющий файлы

```
const fs = require('fs');
const http = require('http');
let server = http.createServer(function(req, res) {
 console.log(req.url);
 res.setHeader('Content-Type', 'text/html; charset=utf-8');
```

```
if (req.url === '/') {
 let stream = fs.createReadStream('index.html');
 stream.pipe(res);
 stream.on('error', err => {
 res.statusCode = 500;
 res.end('<h1>Error</h1>');
 });
}
else {
 res.statusCode = 404;
 res.end('<h1>Not found</h1>');
}
});
server.listen(3000, () => {
 console.log('Сервер запущен на порту 3000');
});
```

Содержимое файла `index.html` (должен находиться в одном каталоге с исполняемым файлом) приведено в листинге 19.5.

#### Листинг 19.5. Файл `index.html`

```
<!DOCTYPE html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>Заголовок</title>
</head>
<body>
 <h1>Содержимое файла</h1>
</body>
</html>
```

## 19.5. Nodemon: автоматическая перезагрузка веб-сервера

После каждого изменения исходного файла необходимо вручную останавливать сервер, а потом запускать его снова — уже исправленную редакцию. Чтобы сервер перезапускался автоматически, нужно установить пакет Nodemon с помощью следующей команды:

```
npm install nodemon --save-dev
```

Внутри файла `package.json` в секцию `devDependencies` будет добавлена зависимость этапа разработки:

```
"nodemon": "^2.0.7"
```

Добавим новый скрипт в секцию `scripts` внутри файла `package.json`:

```
"startServer": "nodemon index.js"
```

Запустим сервер с помощью пакета Nodemon:

```
C:\book\p1>npm run startServer
```

```
> p1@1.0.0 startServer C:\book\p1
```

```
> nodemon index.js
```

```
[nodemon] 2.0.7
```

```
[nodemon] to restart at any time, enter `rs`
```

```
[nodemon] watching path(s): *.*
```

```
[nodemon] watching extensions: js,mjs,json
```

```
[nodemon] starting `node index.js`
```

Сервер запущен на порту 3000

Теперь при изменении файла `index.js` сервер будет автоматически перезагружен, и мы увидим в консоли следующие сообщения:

```
[nodemon] restarting due to changes...
```

```
[nodemon] starting `node index.js`
```

Сервер запущен на порту 3000



## ГЛАВА 20

# Работа JavaScript в веб-браузерах

Начиная с этой главы, мы на время оставим Node.js и начнем изучение особенностей работы JavaScript в веб-браузерах (для которых этот язык и был изначально создан). В последней главе мы вернемся к Node.js, чтобы научиться собирать приложения для веб-браузера.

### ПРИМЕЧАНИЕ

Далее предполагается, что читатель знаком с языками HTML и CSS. Если это не так, то книга "HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера"<sup>1</sup> поможет исправить ситуацию.

## 20.1. Особенности работы JavaScript в веб-браузерах

JavaScript, выполняющийся в веб-браузере, обладает несколькими особенностями. Во-первых, выполняемая программа (*скрипт*) не имеет доступа к каталогам и файлам. Ей доступна только объектная модель документа (под документом имеется в виду веб-страница), веб-браузера и ограниченный доступ в Интернет (по умолчанию загружать данные можно только с того же домена).

Во-вторых, разные веб-браузеры могут по-разному выполнять код программы. Все примеры скриптов в этой книге написаны под Mozilla Firefox и могут не работать в других веб-браузерах. Поэтому на время обучения установите Firefox последней версии.

В-третьих, переменные лучше объявлять с помощью ключевого слова `var`, чтобы программа работала в большинстве веб-браузеров. Глобальные переменные автоматически становятся свойствами глобального объекта `window`. Пример обращения к глобальной переменной через объект `window`:

```
var x = 10;
console.log(window.x); // 10
```

---

<sup>1</sup> Прохоренко Н., Дронов В. HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера. 5-е изд. 2019. 912 с. ISBN 978-5-9775-3986-9.

Проверить существование глобальной переменной можно следующим способом:

```
if (window.str === undefined) {
 console.log('Переменная не существует');
}
else {
 console.log('Переменная существует');
}
```

Обращение к несуществующему свойству объекта не приводит к возникновению ошибки.

Для тестирования программ создадим отдельный каталог p3 (C:\book\p3) и откроем его в редакторе VS Code. Внутри каталога создадим файл test.html в кодировке UTF-8 без BOM. И превратим каталог в пакет, выполнив команду:

```
C:\book\p3>npm init -y
```

## 20.2. Первая JavaScript-программа

При изучении языков программирования принято начинать с программы, выводящей надпись "Hello, world". Не будем нарушать традицию и продемонстрируем, как это будет выглядеть на JavaScript, выполняющемся в веб-браузере (листинг 20.1).

### Листинг 20.1. Первая программа

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>Первая программа</title>
</head>
<body>
<script>
document.write('Hello, world');
</script>
<noscript>
 <p>Ваш веб-браузер не поддерживает JavaScript</p>
</noscript>
</body>
</html>
```

Сохраним код в файле test.html. Запустим Mozilla Firefox, выберем в главном меню пункт **Открыть файл**, в открывшемся окне найдем наш файл test.html и нажмем кнопку **Открыть**.

Если Firefox является веб-браузером по умолчанию, открыть веб-страницу также можно двойным щелчком на значке файла. Если это не так, щелкнем правой кноп-

кой мыши на файле и из контекстного меню выберем пункт **Открыть с помощью | Выбрать программу**. В открывшемся окне найдем Mozilla Firefox и установим флажок **Использовать это приложение для всех файлов html**. Либо в настройках веб-браузера делаем его заданным по умолчанию.

Также можно в адресной строке веб-браузера набрать команду **file:///C:/book/p3/test.html** и нажать клавишу <Enter>.

Возможны следующие варианты:

- ☐ в окне веб-браузера появилась надпись "Hello, world" — все нормально;
- ☐ отобразилась надпись "Ваш веб-браузер не поддерживает JavaScript" — в настройках веб-браузера запрещен запуск скриптов;
- ☐ вообще ничего не появилось — допущена ошибка в коде программы. Следует иметь в виду, что в JavaScript регистр имеет большое значение. Строчные и прописные буквы считаются разными. Более того, важна каждая буква, каждая кавычка. Достаточно ошибиться в одной букве — и программа работать не будет.

Итак, мы столкнулись с первой проблемой при использовании JavaScript — любой пользователь может отключить запуск скриптов в настройках веб-браузера. Но эта проблема не единственная. Разные веб-браузеры могут по-разному выполнять код программы. По этой причине приходится писать персональный код под каждый веб-браузер или пользоваться JavaScript-библиотеками, например jQuery. Все примеры скриптов в этой книге написаны под Firefox версии 86 и могут не работать в других веб-браузерах. Увы, но просто применить Firefox недостаточно, нужно еще учитывать его версию. Если установлена новая версия, то все должно работать, но в более ранних версиях некоторые возможности будут недоступны.

После каждого изменения программы нужно обновить веб-страницу. Обновить в Firefox можно следующими способами:

- ☐ щелкнуть правой кнопкой мыши на содержимом страницы и нажать присутствующий в контекстном меню значок **Обновить текущую страницу**;
- ☐ щелкнуть правой кнопкой мыши на заголовке текущей вкладки и из контекстного меню выбрать пункт **Обновить вкладку**;
- ☐ нажать клавишу <F5> или комбинацию клавиш <Ctrl>+<R>.

Существует возможность автоматического обновления веб-страницы сразу после внесения изменений и сохранения файла. Для этого нужно установить расширение Live Server (см. *разд. 1.7*).

## 20.3. Тег <script>

JavaScript-программа внедряется в код веб-страницы с помощью парного тега <script>. Тег имеет следующие параметры:

- ☐ type — MIME-тип содержимого. Для JavaScript указывается значение text/javascript. Впрочем, параметр является необязательным. Пример:

```
<script type="text/javascript">
document.write('Hello, world');
</script>
```

- ❑ `src` — интернет-адрес файла с программой:

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

- ❑ `defer` — программа, расположенная во внешнем файле, будет выполнена только после полной загрузки веб-страницы:

```
<script src="script.js" defer></script>
```

- ❑ `async` — программа, расположенная во внешнем файле, будет загружена и выполнена асинхронно.

По умолчанию веб-браузер, дойдя до тега `<script>`, приостанавливает загрузку и обработку HTML-кода и приступает к загрузке и обработке JavaScript-программы. Пока программа обрабатывается, пользователь в окне веб-браузера ничего не будет видеть. Параметр `async` предписывает веб-браузеру обрабатывать HTML-код и JavaScript-программу одновременно. Пример:

```
<script src="script.js" async></script>
```

Если выполнение скриптов запрещено в настройках веб-браузера, будет выведен текст между тегами `<noscript>` и `</noscript>`.

## 20.4. Расположение JavaScript-программы

В листинге 20.1 мы поместили тег `<script>` внутри тега `<body>` и для вывода надписи "Hello, world" воспользовались методом `write()` встроенного объекта `document`:

```
document.write('Hello, world');
```

Иногда JavaScript-программа помещается в теге `<head>` (листинг 20.2).

### Листинг 20.2. Размещение программы в теге `<head>`

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>Размещение программы в теге head</title>
 <script>
 function printHelloWorld() {
 var div1 = document.getElementById('div1');
 div1.innerHTML = 'Hello, world';
 }
 </script>
</head>
<body onload="printHelloWorld()">
 <div id="div1">Ваш веб-браузер не поддерживает JavaScript</div>
```



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

В этом случае пользоваться методом `write()` нельзя, т. к. выведенный им текст окажется помещенным в тег `<head>`, содержимое которого не отображается на экране. Придется прибегнуть к другому способу.

Сначала необходимо решить, в каком элементе страницы должен выводиться текст, и дать этому элементу уникальный идентификатор. Идентификатор задается непосредственно в HTML-теге, формирующем элемент, в параметре `id`. Например:

```
<div id="div1">Ваш веб-браузер не поддерживает JavaScript</div>
```

Далее необходимо получить объект, представляющий данный элемент. Для этого следует вызвать метод `getElementById()` объекта `document`, передав методу идентификатор нужного элемента. Вот так:

```
var div1 = document.getElementById('div1');
```

Останется лишь присвоить выводимый текст свойству `innerHTML` полученного объекта. Новый текст заменит старый, присутствовавший в этом элементе ранее. Пример:

```
div1.innerHTML = 'Hello, world';
```

Следует также учитывать, что по умолчанию код программы, расположенный в теге `<head>`, выполняется раньше, чем сформируется структура страницы. В итоге мы будем получать ссылку на элемент, которого еще не существует.

Избежать этого можно, оформив код как обработчик события `onload` тега `<body>`, которое возникает сразу после загрузки страницы.

Мы оформили код, выводящий текст на экран, в виде функции `printHelloWorld()` и присвоили фрагмент кода, содержащий вызов этой функции, параметру `onload` тега `<body>`, тем самым назначив упомянутую ранее функцию обработчиком события `onload`:

```
<body onload="printHelloWorld()">
```

Теперь код выполнится после загрузки страницы, когда элемент с идентификатором `div1` уже будет существовать, и текст будет выведен без проблем.

JavaScript-программа может быть помещена в конце секции тела страницы, перед закрывающим тегом `<body>`:

```
<html lang="ru">
. . .
<body>
 <div id="div1">Ваш веб-браузер не поддерживает JavaScript</div>
 <script>
 var div1 = document.getElementById('div1');
 div1.innerHTML = 'Hello, world';
 </script>
```

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

В таком случае JavaScript-программа выполняется после загрузки содержимого страницы, когда все элементы уже сформированы и к ним можно без проблем получить доступ из программного кода.

Сложную программу удобнее вынести в отдельный файл с расширением js (листинг 20.3). Интернет-адрес такого файла указывается в параметре `src` тега `<script>` (листинг 20.4).

#### Листинг 20.3. Содержимое файла `script.js`

```
function printHelloWorld() {
 var div1 = document.getElementById('div1');
 div1.innerHTML = 'Hello, world';
}
```

#### Листинг 20.4. Размещение программы в отдельном файле

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>Размещение программы в отдельном файле</title>
 <script src="script.js"></script>
</head>
<body onload="printHelloWorld()">
 <div id="div1">Ваш веб-браузер не поддерживает JavaScript</div>
</body>
</html>
```

Если в теге `<script>` указан параметр `defer` (листинг 20.5), то код из файла `script.js` (листинг 20.6) будет выполнен только после полной загрузки страницы (как если бы код программы был помещен после закрывающего тега `<html>`). В этом случае оформлять код в виде обработчика события `onload` не нужно.

#### Листинг 20.5. Использование параметра `defer` тега `<script>`

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>Параметр defer тега script</title>
 <script src="script.js" defer></script>
</head>
<body>
 <div id="div1">Ваш веб-браузер не поддерживает JavaScript</div>
```

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

**Листинг 20.6. Содержимое файла script.js**

```
var div1 = document.getElementById('div1');
div1.innerHTML = 'Hello, world';
```

## 20.5. Расположение определений функций в HTML-коде

Обычно код с определениями функций помещают в теге `<head>` страницы (листинг 20.7) или в отдельном файле с расширением `js` (листинги 20.8 и 20.9).

**Листинг 20.7. Функция расположена в теге `<head>`**

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>Функции</title>
 <script>
 function sum(x, y) {
 return x + y;
 }
 </script>
</head>
<body>
 <script>
 var var1 = 5, var2 = 2, var3;
 var3 = sum(var1, var2);
 document.write('var3 = ' + var3);
 </script>
</body>
</html>
```

**Листинг 20.8. Функция вынесена в отдельный файл script.js**

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>Функции</title>
 <script src="script.js"></script>
</head>
```

```
<body>
 <script>
 let var1 = 5, var2 = 2, var3;
 var3 = sum(var1, var2);
 document.write('var3 = ' + var3);
 </script>
</body>
</html>
```

**Листинг 20.9. Содержимое файла script.js**

```
function sum(x, y) {
 return x + y;
}
```

## 20.6. Консоль Mozilla Firefox

Веб-браузер Firefox содержит панель **Инструменты разработчика**. В состав этого средства входит **Веб-консоль** (рис. 20.1), в которую выводятся различные сообщения (например, об ошибках в программе). Чтобы открыть консоль, следует перейти в главное меню и выбрать пункт **Веб-разработка | Консоль браузера** или нажать комбинацию клавиш **<Ctrl>+<Shift>+<J>**.

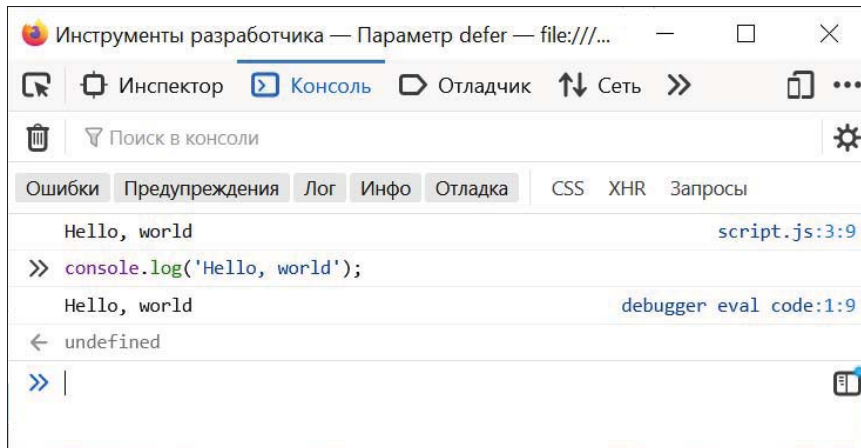


Рис. 20.1. Консоль Mozilla Firefox

В верхней части вкладки **Консоль** расположены переключатели **Ошибки**, **Предупреждения**, **Лог**, **Инфо**, **Отладка**, **CSS**, **XHR** и **Запросы**, с помощью которых можно включить или отключить вывод сообщений разных типов. Убедитесь, что все переключатели подсвечены серым цветом, т. е. установлены (если фон переключателя белого цвета, то переключатель сброшен и вывод сообщений соответствующего типа отключен). Держите консоль всегда открытой при написании и отладке программы — и сразу заметите наличие ошибок в программе.

Консоль также можно использовать для вывода произвольных сообщений (например, с отладочной информацией). Для этого можно использовать, например, следующий код:

```
<script>
 console.log('Hello, world');
</script>
```

Сообщение "Hello, world" отобразится в консоли, если переключатель **Лог** установлен.

В консоли можно набирать инструкции JavaScript — и они будут исполняться в контексте текущей веб-страницы (см. рис. 20.1). В том числе можно получать и изменять параметры элементов этой страницы. Очередная выполняемая инструкция вводится в строке, помеченной приглашением `>>`, после чего следует нажать клавишу `<Enter>`. Пример ввода инструкций и получения результата:

```
>> console.log('Hello, world');
 Hello, world
<- undefined
```

Вторая строка была выведена выполнившейся инструкцией, а третья строка — это значение, возвращаемое методом `log()` объекта `console`.

Также представляет интерес вкладка **Отладчик**, которая открывается при возникновении ошибки в исполняющемся коде. На этой вкладке существует возможность выполнять программу по шагам, отслеживая значения различных переменных. Очень удобное средство для поиска ошибок в программе, особенно логических.

## 20.7. Встроенные диалоговые окна

Встроенные диалоговые окна позволяют выводить всевозможные сообщения, результаты работы программы и вводить исходные данные.

### 20.7.1. Окно с сообщением и кнопкой **ОК**

Метод `alert()` объекта `window` отображает диалоговое окно с заданным сообщением и кнопкой **ОК**. В листинге 20.10 демонстрируется вывод приветствия с помощью этого метода.

#### Листинг 20.10. Метод `alert()`

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>Метод alert()</title>
</head>
```

```
<body>
<script>
 window.alert('Hello, world');
</script>
</body>
</html>
```

Объект `window` перед методом `alert()` указывать необязательно (поскольку код и так выполняется в контексте этого объекта):

```
alert('Hello, world');
```

Сообщение можно разбить на строки с помощью последовательности символов `\n`:

```
alert('Строка1\nСтрока2\n\nСтрока4');
```

### 20.7.2. Окно с сообщением и кнопками **ОК** и **Отмена**

Метод `confirm()` объекта `window` отображает диалоговое окно с заданным сообщением и кнопками **ОК** и **Отмена** (листинг 20.11). Он возвращает значение `true`, если пользователь нажал кнопку **ОК**, и `false` — если нажал кнопку **Отмена** или просто закрыл окно щелчком на кнопке закрытия в заголовке.

#### Листинг 20.11. Метод `confirm()`

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>Метод confirm()</title>
</head>
<body>
<script>
if (window.confirm('Нажмите одну из кнопок')) {
 alert('Нажата кнопка ОК');
}
else {
 alert('Нажата кнопка Отмена');
}
</script>
</body>
</html>
```

### 20.7.3. Окно с полем ввода и кнопками **ОК** и **Отмена**

Метод `prompt()` объекта `window` отображает диалоговое окно с заданным сообщением, полем ввода и кнопками **ОК** и **Отмена** (листинг 20.12). Он возвращает введенное значение, если была нажата кнопка **ОК**, или `null`, если была нажата кнопка **Отмена** или окно было закрыто щелчком на кнопке в заголовке.

**Листинг 20.12. Метод `prompt()`**

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>Метод prompt()</title>
</head>
<body>
<script>
var n = prompt('Введите ваше имя', 'Это значение по умолчанию');
if (n === null) {
 document.write('Вы нажали Отмена');
}
else {
 document.write('Привет ' + n);
}
</script>
</body>
</html>
```

## 20.8. Получение доступа к элементам веб-страницы

Получить доступ к объекту, представляющему нужный элемент страницы, позволяют следующие свойства и методы объекта `document`:

- ❑ `body` — ссылка на тег `<body>`;
- ❑ `activeElement` — ссылка на элемент формы, имеющий фокус ввода, или `null`, если ни один элемент формы не имеет фокуса ввода;
- ❑ `getElementById(<Идентификатор>)` — возвращает ссылку на элемент с указанным идентификатором или значение `null`, если элемента с таким идентификатором нет:

```
var div1 = document.getElementById('div1');
```

- ❑ `getElementsByName(<Название элемента>)` — возвращает ссылку на коллекцию элементов с заданным названием (название элемента задается в параметре `name`). Если ни одного элемента с указанным названием нет, возвращает пустую коллекцию;
- ❑ `getElementsByTagName(<Тег>)` — возвращает ссылку на коллекцию дочерних элементов, созданных с использованием тега, имя которого передано в качестве параметра (указывается без символов `<` и `>`). Если ни одного подходящего элемента нет, возвращает пустую коллекцию;
- ❑ `getElementsByClassName(<Стилевой класс>)` — возвращает ссылку на коллекцию элементов, для которых задан указанный стилевой класс (указывается без точки

в начале). Если ни одного подходящего элемента нет, возвращает пустую коллекцию. Пример:

```
<div id="div1" class="cls1"></div>
. . .
var arr = document.getElementsByClassName('cls1');
console.log(arr[0].id); // div1
```

- ❑ `querySelector(<CSS-селектор>)` — возвращает ссылку на первый элемент, соответствующий указанному CSS-селектору, или значение `null`, если подходящего элемента нет:

```
<div id="div1" class="cls1"></div>
. . .
console.log(document.querySelector('.cls1').id); // div1
console.log(document.querySelector('#div1').className); // cls1
```

- ❑ `querySelectorAll(<CSS-селектор>)` — возвращает ссылку на коллекцию элементов, соответствующих указанному CSS-селектору. Если ни одного подходящего элемента нет, возвращает пустую коллекцию. Получим содержимое всех абзацев:

```
<p>Абзац 1</p><p>Абзац 2</p>
. . .
var arr = document.querySelectorAll('p');
for (var i = 0; i < arr.length; i++) {
 console.log(arr[i].innerHTML);
}
```

## 20.9. Создание часов на веб-странице

В листинге 20.13 приведен код веб-страницы с часами, отображающими время вплоть до секунды, кнопками для запуска и остановки часов. Часы реализованы посредством таймера (см. *разд. 9.7*).

### Листинг 20.13. Часы на веб-странице

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>Часы на веб-странице</title>
<script>
let timerId;
function numToStr(value) {
 value += ''; // Преобразуем число в строку
 if (value.length == 1) return '0' + value;
 else return value;
}
```



```

function startTimer() { // Запускаем таймер
 timerId = setInterval(function() {
 let d = new Date();
 let msg = numToStr(d.getHours()) + ':';
 msg += numToStr(d.getMinutes()) + ':';
 msg += numToStr(d.getSeconds());
 document.getElementById('div1').innerHTML = msg;
 }, 1000);
 document.getElementById('div_start').style.display = 'none';
 document.getElementById('div_stop').style.display = 'block';
}

function stopTimer() { // Останавливаем таймер
 clearInterval(timerId);
 document.getElementById('div_start').style.display = 'block';
 document.getElementById('div_stop').style.display = 'none';
}

</script>
</head>
<body onload="startTimer()">
<div id="div1"></div>
<div id="div_start">
 <input type="button" value="Запустить часы" onclick="startTimer()">
</div>
<div id="div_stop">
 <input type="button" value="Остановить часы" onclick="stopTimer()">
</div>
</body>
</html>

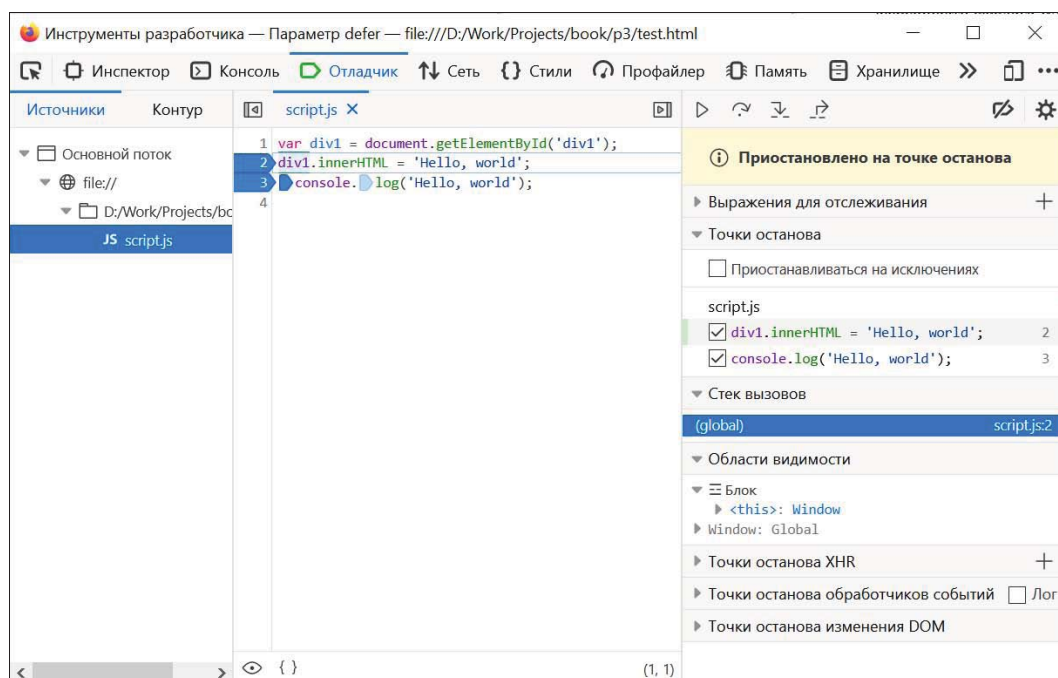
```

## 20.10. Отладчик Mozilla Firefox

Веб-браузер Firefox в составе Инструментов разработчика содержит не только консоль (см. *разд. 20.6*), но и полноценный отладчик (рис. 20.2). Чтобы вывести его, следует перейти в главное меню, выбрать пункт **Веб-разработка | Инструменты веб-разработчика** (или нажать комбинацию клавиш <Ctrl>+<Shift>+<I>) и переключиться на вкладку **Отладчик**.

Панель отладчика состоит из трех частей. В левой части выводится своего рода Проводник, в котором можно выбрать файл с отлаживаемым кодом: HTML-файл, содержащий код на JavaScript, или JS-файл. В средней части выводится содержимое выбранных в левой части файлов. Если было открыто несколько файлов, они будут выведены на отдельных вкладках, подобно VS Code. Наконец, в правой части отображается всевозможная отладочная информация, в частности перечень созданных точек останова (были описаны в *разд. 13.7*).

Чтобы установить точку останова, достаточно щелкнуть мышью в средней части отладчика, левее нужной строки кода, на полосе, в которой выводятся номера строк. Повторный щелчок на точке останова убирает ее.

Рис. 20.2. Пошаговое выполнение программы на вкладке **Отладчик**

Все установленные точки останова перечисляются в графе **Точки останова** правой части отладчика. Каждая позиция, представляющая точку останова, включает в себя флажок, сбросив который можно временно отключить точку останова (установка флажка вновь включает ее), строку программного кода, на которой стоит точка останова, и порядковый номер этой строки.

После того как выполнение программы будет остановлено по достижении точки останова, можно просмотреть текущее значение любой переменной, созданной в коде. Для этого достаточно навести курсор мыши на имя переменной — и ее значение отобразится в небольшом всплывающем окошке.

Также можно выполнить программу по шагам, пользуясь следующими кнопками, которые расположены на панели сверху правой части отладчика:

- **Возобновить** (эквивалент — клавиша <F8>) — выполняет программу до следующей точки останова или до конца, если далее точек останова нет;
- **Перешагнуть через** (клавиша <F10>) — выполняет одну инструкцию. Если в инструкции осуществляется вызов функции, то функция выполняется и отладчик переходит в режим ожидания после выхода из функции;
- **Зайти в** (клавиша <F11>) — выполняет одну инструкцию. Если в инструкции есть вызов функции, то функция также будет выполняться пошагово;
- **Выйти из** (комбинация клавиш <Shift>+<F11>) — позволяет сразу выйти из функции;

- ❑ **Отключить точки останова** — временно отключает все точки останова;
- ❑ **Включить точки останова** — включает все ранее отключенные точки останова.

## 20.11. JavaScript-библиотеки

Разные веб-браузеры могут по-разному выполнять код программы. По этой причине при написании приложений приходится учитывать особенности каждого веб-браузера. Проблема заключается в том, что установить все версии каждого веб-браузера на один компьютер практически невозможно, а значит, обеспечить полную кросс-браузерность самостоятельно не получится.

Ряд существующих JavaScript-библиотек предоставляют удобные в использовании кросс-браузерные инструменты, предназначенные для выполнения различных действий над веб-страницей и ее элементами. Вот наиболее распространенные библиотеки такого рода:

- ❑ jQuery — <https://jquery.com/>;
- ❑ Prototype — <http://prototypejs.org/>;
- ❑ MooTools — <https://mootools.net/>;
- ❑ Dojo — <https://dojotoolkit.org/>;
- ❑ Yahoo! UI Library (YUI) — <https://yuilib.com/>.

Из этого списка следует особо выделить библиотеку jQuery. Она удобна, компактна и не засоряет глобальное пространство имен обилием переменных, констант, функций и классов. Большой популярности jQuery способствовало наличие множества дополнительных модулей, добавляющих библиотеке новую функциональность. Например, дополнительный модуль jQuery UI предоставляет готовые компоненты: "аккордеон", панель с вкладками, диалоговые окна, календарь и др.

Подключим библиотеку jQuery и выведем с ее помощью приветствие (листинг 20.14). Предварительно нужно скачать библиотеку со страницы <https://jquery.com/download/>. Скачиваем файл, например, jquery-3.6.0.min.js (он содержит компактную редакцию библиотеки без комментариев, лишних пробельных символов и рекомендуемую для применения разработчиками JavaScript-программ), размещаем его в одной папке с веб-страницей и для удобства переименовываем в jquery.js.

### Листинг 20.14. Вывод приветствия с помощью библиотеки jQuery

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>Вывод приветствия с помощью библиотеки jQuery</title>
 <script src="jquery.js"></script>
 <script>
```

```
$(function() {
 $('#div1').text('Hello, world');
});
</script>
</head>
<body>
 <div id="div1"></div>
</body>
</html>
```

Этот код аналогичен коду из листинга 20.2, только проще. В теге `<head>` подключаем библиотеку jQuery. Затем создаем анонимную функцию, которая будет вызвана, когда структура страницы будет сформирована, причем еще до окончания загрузки других элементов страницы, например изображений.

Всю функциональность в библиотеке jQuery выполняет функция `$()`. Она имеет псевдоним `jQuery()`:

```
jQuery(function() {
 jQuery('#div1').text('Hello, world');
});
```

Функция `$()` имеет несколько форматов. Наиболее часто применяемый:

```
$(<CSS-селектор>)
```

В листинге 20.14 мы получили доступ к элементу по его идентификатору:

```
$('#div1').text('Hello, world');
```

Функция возвращает ссылку на один элемент или на целый их набор. Используя различные методы, можно выполнять всевозможные операции над элементами. Если функция возвращает набор элементов, то операции в большинстве случаев выполняются над всеми элементами из набора. Например, выведем текст приветствия во всех абзацах:

```
<p></p>
<p></p>
<p></p>
<script>
$(function() {
 $('p').text('Hello, world');
});
</script>
```

Чтобы вывести текст только в первом абзаце, достаточно добавить метод `first()`:

```
$('p').first().text('Hello, world');
```

Существуют и более сложные, высокоуровневые JavaScript-библиотеки — фреймворки, реализующие более развитую функциональность. К числу таких фреймворков относятся, в частности, React, Vue.js и Angular.



## ГЛАВА 21

# Объектная модель документа

*Объектная модель документа* (Document Object Model, DOM) — это совокупность объектов, представляющих саму веб-страницу, ее отдельные элементы и ряд подсистем веб-браузера.

## 21.1. Структура объектной модели документа

Объектная модель документа имеет иерархическую структуру. Существует объект верхнего уровня, ряд подчиненных ему объектов, которым, в свою очередь, подчинены другие объекты, и т. д.

Объектом верхнего уровня является `window`, представляющий текущее окно веб-браузера. Этот объект содержит метод `alert()`, выводящий окно с заданной надписью и кнопкой **ОК**. Пример:

```
window.alert('Сообщение');
```

В объект `window` вложен объект `document`, представляющий веб-страницу, открытую в текущем окне. Этот объект содержит метод `write()`, выводящий на страницу заданный текст. Пример:

```
window.document.write('Сообщение');
```

Поскольку весь программный код выполняется в контексте объекта `window`, этот объект можно не указывать:

```
alert('Сообщение');
```

```
document.write('Сообщение');
```

Объекту `window` также подчинены следующие основные объекты:

- ☐ `frames` — коллекция фреймов (см. *разд. 21.3*);
- ☐ `navigator` — сведения о веб-браузере (см. *разд. 21.4*);
- ☐ `screen` — сведения об экране клиентского компьютера (см. *разд. 21.5*);
- ☐ `location` — интернет-адрес текущей веб-страницы (см. *разд. 21.6*);
- ☐ `history` — история веб-браузера (см. *разд. 21.7*);

- `document` — текущая страница. Этому объекту подчинены следующие:
  - `forms` — коллекция всех форм;
  - `images` — коллекция всех изображений;
  - `links` — коллекция всех гиперссылок;
  - `scripts` — коллекция всех скриптов.

## 21.2. Объект *window*

Как уже упоминалось, объект `window` — это объект самого верхнего уровня, представляющий текущее окно веб-браузера. Весь программный код выполняется в контексте объекта `window`, поэтому ссылку на него при написании кода указывать не обязательно.

Свойства объекта `window`:

- `screenX` и `screenLeft` — горизонтальная координата левого верхнего угла окна;
- `screenY` и `screenTop` — вертикальная координата левого верхнего угла окна;
- `outerWidth` — полная ширина окна в пикселах;
- `outerHeight` — полная высота окна в пикселах;
- `innerWidth` — ширина внутренней области окна в пикселах (за вычетом толщины рамок окна и ширины служебных панелей, однако с учетом ширины вертикальной полосы прокрутки, если она присутствует);
- `innerHeight` — высота внутренней области окна в пикселах (за вычетом толщины рамок, высоты строки меню и служебных панелей, однако с учетом высоты горизонтальной полосы прокрутки, если она присутствует);
- `scrollX` и `pageXOffset` — текущая позиция горизонтальной прокрутки в пикселах;
- `scrollY` и `pageYOffset` — текущая позиция вертикальной прокрутки в пикселах.

Методы объекта `window`:

- `alert()` — отображает окно сообщения (см. *разд. 20.7.1*);
- `confirm()` — выводит окно подтверждения (см. *разд. 20.7.2*);
- `prompt()` — показывает окно с полем ввода (см. *разд. 20.7.3*);
- `scrollTo(<X>, <Y>)` — прокручивает содержимое окна в точку с координатами `<X>` и `<Y>`;
- `scrollBy(<tX>, <tY>)` — прокручивает окно на значение параметра `<tX>` по горизонтали и на значение параметра `<tY>` по вертикали;
- `open()` — открывает новое окно.

Кроме того, имеются четыре метода для работы с таймерами, которые мы рассмотрели в *разд. 9.7*.

## 21.3. Работа с фреймами

Для работы с фреймами предназначены следующие свойства объекта `window`:

- `frames` — коллекция фреймов. Обратиться к элементам коллекции можно по индексу, указанному внутри квадратных скобок, а также по названию (значению параметра `name` тега `<iframe>`). Общее число фреймов доступно через свойство `length`. Пример:

```
console.log(window.frames.length);
console.log(window.frames[0].document.title);
console.log(window.frames.frame1.document.title);
```

- `length` — число фреймов, отображаемых в текущем окне;

- `name` — имя окна или фрейма;

- `parent` — ссылка на родительское окно или фрейм:

```
console.log('parent: ' + window.parent.document.title);
```

- `self` и `window` — ссылка на текущее окно или фрейм:

```
console.log('self: ' + window.self.document.title);
```

- `top` — ссылка на окно (используется, чтобы получить доступ к окну из фрейма):

```
console.log('top: ' + window.top.document.title);
```

### ОБРАТИТЕ ВНИМАНИЕ

В целях безопасности страница, загруженная с другого домена, не имеет доступа к родительскому окну.

## 21.4. Объект *navigator*: получение сведений о веб-браузере

Объект `navigator` предоставляет сведения о самом веб-браузере.

Свойства объекта `navigator`:

- `userAgent` — строка, содержащая описание веб-браузера и клиентской операционной системы (пользователь в настройках может изменить эту информацию):

```
console.log(window.navigator.userAgent);
// Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:88.0)
// Gecko/20100101 Firefox/88.0
```

- `language` — строковое обозначение предпочитаемого пользователем языка:

```
console.log(navigator.language); // ru-RU
```

- `languages` — массив со строковыми обозначениями языков в порядке предпочтения:

```
console.log(navigator.languages);
// Array(4) ["ru-RU", "ru", "en-US", "en"]
```

- ❑ `onLine` — `true`, если клиент в настоящее время подключен к сети, и `false`, если отключен:

```
console.log(navigator.onLine); // true
```

- ❑ `cookieEnabled` — `true`, если прием `cookie` разрешен, `false` — если запрещен в настройках веб-браузера:

```
console.log(navigator.cookieEnabled); // true
```

## 21.5. Объект *screen*: получение сведений об экране клиентского компьютера

Объект `screen` содержит сведения о характеристиках экрана клиентского компьютера.

Свойства объекта `screen`:

- ❑ `width` — полная ширина экрана в пикселах;

- ❑ `height` — полная высота экрана в пикселах:

```
console.log(window.screen.width); // 2560
console.log(window.screen.height); // 1440
```

- ❑ `availWidth` — ширина свободной части экрана в пикселах (за вычетом ширины служебных панелей);

- ❑ `availHeight` — высота свободной части экрана в пикселах (за вычетом высоты служебных панелей):

```
console.log(screen.availWidth); // 2560
console.log(screen.availHeight); // 1400
```

- ❑ `colorDepth` — глубина цвета (число битов, отводимых на кодирование цвета одного экранного пиксела):

```
console.log(screen.colorDepth); // 24
```

## 21.6. Объект *location*: доступ к интернет-адресу веб-страницы

Объект `location` содержит информацию об интернет-адресе текущей страницы.

Свойства объекта `location`:

- ❑ `href` — полный интернет-адрес страницы (доступно для чтения и записи):

```
console.log(window.location.href);
```

- ❑ `protocol` — обозначение протокола с двоеточием (например, `'http: '`);

- ❑ `hostname` — адрес хоста;



- ❑ `host` — адрес хоста с номером TCP-порта или один лишь адрес хоста, если порт в интернет-адресе не указан;
- ❑ `port` — номер TCP-порта или пустая строка, если порт не указан;
- ❑ `pathname` — путь к запрашиваемому файлу;
- ❑ `search` — строка GET-параметров с начальным символом `?` или пустая строка, если GET-параметров нет;
- ❑ `hash` — якорь (задаваемый параметром `id`) с начальным символом `#` или пустая строка, если якоря нет.

Методы объекта `location`:

- ❑ `assign(<Интернет-адрес>)` — загружает страницу с заданным интернет-адресом;
- ❑ `replace(<Интернет-адрес>)` — загружает страницу с заданным интернет-адресом. Интернет-адрес ранее открытой страницы будет удален из истории веб-браузера;
- ❑ `reload([<Флаг>])` — перезагружает страницу. Если в параметре указано значение `true`, страница будет перезагружена с сервера, а если `false` (или параметр не указан) — то по возможности из кэша.

Загрузить новую страницу можно не только с помощью методов `assign()` или `replace()`, но и присвоив новый интернет-адрес свойству `href` объекта `location`:

```
location.href = 'newURL.html';
```

## 21.7. Объект *history*: история веб-браузера

Объект `history` предоставляет доступ к истории веб-браузера — списку всех просмотренных за последнее время страниц.

Свойство `length` объекта `history` возвращает размер списка истории:

```
console.log(window.history.length);
```

Методы объекта `history`:

- ❑ `back()` — выводит предыдущую страницу из списка истории;
- ❑ `forward()` — выводит следующую страницу из списка истории;
- ❑ `go(<Позиция>)` — выводит страницу, расположенную на позиции списка истории, относительный номер которой указан в качестве параметра (можно указать как положительное, так и отрицательное значение).

## 21.8. Объект *document*: работа с веб-страницей

Объект `document` предоставляет доступ к самой странице.

Свойства объекта `document`:

- ❑ `title` — название страницы, указанное в теге `<title>` (доступно для чтения и записи):

```
document.title = 'Новое название';
console.log(document.title); // Новое название
```

- ❑ URL — интернет-адрес страницы в виде строки;
- ❑ `referrer` — интернет-адрес, с которого посетитель перешел на текущую страницу, в виде строки;
- ❑ `documentElement` — ссылка на корневой элемент (тег `<html>`);
- ❑ `head` — ссылка на тег `<head>`;
- ❑ `body` — ссылка на тег `<body>`;
- ❑ `activeElement` — ссылка на элемент формы, в данный момент имеющий фокус ввода, или `null`, если ни один элемент формы не имеет фокуса ввода;
- ❑ `readyState` — статус загрузки страницы. Может хранить следующие строковые значения:
  - `'loading'` — страница загружается;
  - `'interactive'` — страница загружена не полностью, но доступна для просмотра и управления;
  - `'complete'` — страница загружена полностью;
- ❑ `visibilityState` — статус видимости страницы. Может хранить следующие строковые значения:
  - `'visible'` — страница видима по крайней мере частично (например, если она открыта в окне, выведенном на передний план или частично перекрытом другим окном);
  - `'hidden'` — страница невидима (например, открыта в минимизированном окне или окне, полностью скрытом другим);
- ❑ `lastModified` — дата и время последнего изменения файла страницы в виде строки:

```
console.log(document.lastModified); // "05/25/2021 05:28:34"
```
- ❑ `cookie` — объект `cookie`, который позволяет сохранять данные на компьютере клиента.

Методы объекта `document`:

- ❑ `getElementById(<Идентификатор>)` — возвращает ссылку на элемент с указанным идентификатором или значение `null`, если элемента с таким идентификатором нет:

```
var div1 = document.getElementById('div1');
```
- ❑ `getElementsByName(<Название элемента>)` — возвращает ссылку на коллекцию элементов с заданным названием (которое указывается в параметре `name`) или пустую коллекцию, если ни одного элемента с указанным названием нет;
- ❑ `getElementsByTagName(<Тег>)` — возвращает ссылку на коллекцию дочерних элементов, созданных с использованием тега, имя которого указано в параметре

(задается без символов `<` и `>`). Если ни одного подходящего элемента нет, возвращает ссылку на пустую коллекцию;

- `getElementsByClassName(<Стилевой класс>)` — возвращает ссылку на коллекцию элементов, для которых задан указанный стилевой класс (указывается без точки в начале). Если ни одного подходящего элемента нет, возвращает ссылку на пустую коллекцию. Пример:

```
<div id="div1" class="cls1"></div>
. . .
var arr = document.getElementsByClassName('cls1');
console.log(arr[0].id); // div1
```

- `querySelector(<CSS-селектор>)` — возвращает ссылку на первый элемент, соответствующий указанному CSS-селектору, или значение `null`, если подходящего элемента нет:

```
<div id="div1" class="cls1"></div>
. . .
console.log(document.querySelector('.cls1').id); // div1
console.log(document.querySelector('#div1').className); // cls1
```

- `querySelectorAll(<CSS-селектор>)` — возвращает ссылку на коллекцию элементов, соответствующих указанному CSS-селектору. Если ни одного подходящего элемента нет, возвращает пустую коллекцию. Получим содержимое всех абзацев:

```
<p>Абзац 1</p><p>Абзац 2</p>
. . .
var arr = document.querySelectorAll('p');
for (var i = 0; i < arr.length; i++) {
 console.log(arr[i].innerHTML);
}
```

- `elementFromPoint(<X>, <Y>)` — возвращает ссылку на элемент, находящийся по координатам `<X>` и `<Y>`, или значение `null`, если по заданным координатам ничего нет;

- `write(<HTML-код>)` — вставляет заданный HTML-код в текущее место страницы:
 

```
document.write('<p>Абзац</p>');
```

- `writeln(<HTML-код>)` — вставляет заданный HTML-код в текущее место страницы, а в конце строки добавляет символы возврата каретки и перевода строки. Фактически полностью аналогичен методу `write()`.

Методы `write()` и `writeln()` обычно используются при формировании содержимого страницы на этапе загрузки.

Коллекции объекта `document`:

- `forms` — коллекция всех форм;
- `images` — коллекция всех изображений;

- `links` — коллекция всех гиперссылок;
- `scripts` — коллекция всех скриптов.

Получить число элементов (размер) любой коллекции позволяет свойство `length`. Доступ к элементам коллекции осуществляется по индексу внутри квадратных скобок. Получим интернет-адреса всех гиперссылок:

```
var arr = document.links;
for (var i = 0; i < arr.length; i++) {
 console.log(arr[i].href);
}
```

## 21.9. Узлы DOM

Узел DOM — это объект, представляющий какой-либо элемент страницы: тег, фрагмент текста, находящийся в теге, или комментарий.

Объект-узел предоставляет следующие свойства:

- `nodeType` — целочисленный тип узла. Может принимать следующие значения:
  - 1 — `ELEMENT_NODE` — тег;
  - 3 — `TEXT_NODE` — простой текст;
  - 8 — `COMMENT_NODE` — комментарий;
  - 9 — `DOCUMENT_NODE` — веб-страница;
- `nodeName` — имя узла в виде строки: у тега — имя тега, у текста — `'#text'`, у комментария — `'#comment'`, у страницы — `'#document'`;
- `nodeValue` — значение узла: у текста — собственно сам текст, у комментария — содержимое комментария, у узлов остальных типов — `null`;
- `childNodes` — список всех узлов, вложенных в текущий узел (*дочерних*, или *потомков*):

```
var arr = document.body.childNodes;
for (var i = 0; i < arr.length; i++) {
 console.log(arr[i].nodeType);
 console.log(arr[i].nodeName);
 console.log(arr[i].nodeValue);
}
```

- `firstChild` — первый дочерний узел или `null`, если такого узла нет;
- `lastChild` — последний дочерний узел или `null`, если узла нет;
- `parentNode` — узел, в который непосредственно вложен текущий узел (*родительский*, или *родитель*), или `null`, если такого нет;
- `previousSibling` — узел, непосредственно предшествующий текущему узлу, или `null`, если такого нет;

- `nextSibling` — узел, непосредственно следующий за текущим узлом, или `null`, если такого нет.

Создать новый узел и добавить его в содержимое страницы позволяют следующие методы:

- `createElement(<Имя тега>)` — создает новый узел-тег с заданным именем. Вызывается у объекта `document`;
- `createTextNode(<Текст>)` — создает новый узел-текст. Вызывается у объекта `document`;
- `appendChild(<Новый узел>)` — добавляет новый узел в конец содержимого текущего узла. Если узел уже находится в содержимом страницы, то удаляет его и вставляет в текущий узел. Возвращает ссылку на добавленный узел. В качестве примера создадим абзац с гиперссылкой и добавим его в конец страницы:

```
var p = document.createElement('p'); // Создаем абзац
var text = document.createTextNode('Это текст абзаца ');
p.appendChild(text); // Добавляем текст в абзац
var link = document.createElement('a'); // Создаем гиперссылку
link.href = 'test.html'; // Задаем интернет-адрес гиперссылки
var linkText = document.createTextNode('Это текст гиперссылки');
link.appendChild(linkText); // Добавляем текст в гиперссылку
p.appendChild(link); // Добавляем гиперссылку в конец абзаца
// Добавляем новый узел в конец страницы
document.body.appendChild(p);
```

- `insertBefore(<Новый узел>, <Узел>)` — добавляет заданный новый узел в текущий перед узлом, указанным вторым параметром. Если вторым параметром передать `null`, новый узел будет помещен в конец содержимого текущего узла. Если узел уже находится в содержимом страницы, то удаляет его и вставляет в текущий узел. Возвращает ссылку на вставленный узел. Пример:

```
var p = document.getElementsByTagName('p')[0];
// Перемещаем первый абзац в конец страницы
document.body.insertBefore(p, document.body.lastChild);
```

**Вставим новый абзац в начало страницы:**

```
var p = document.createElement('p'); // Создаем абзац
var text = document.createTextNode('Это текст абзаца ');
p.appendChild(text); // Добавляем текст в абзац
// Добавляем новый узел в начало страницы
document.body.insertBefore(p, document.body.firstChild);
```

- `cloneNode(true | false)` — создает и возвращает копию текущего узла. Если в качестве параметра указано значение `true`, то будут скопированы все потомки данного узла. Обработчики событий не копируются. Сделаем полную копию абзаца и добавим новый узел в конец страницы:

```
var p = document.getElementsByTagName('p')[0].cloneNode(true);
// Добавляем в конец страницы
document.body.appendChild(p);
```

- ❑ `removeChild(<Удаляемый узел>)` — удаляет заданный дочерний узел из текущего узла и возвращает ссылку на удаленный узел. В качестве примера удалим первый абзац:

```
var p = document.getElementsByTagName('p')[0];
p.parentNode.removeChild(p);
```

- ❑ `replaceChild(<Новый узел>, <Старый узел>)` — заменяет указанный старый узел, являющийся потомком текущего узла, заданным новым узлом. Возвращает ссылку на замененный старый узел. Заменим первый тег `<div>` на новый абзац:

```
var p = document.createElement('p'); // Создаем абзац
var text = document.createTextNode('Это текст абзаца');
p.appendChild(text); // Добавляем текст в абзац
var div = document.getElementsByTagName('div')[0];
div.parentNode.replaceChild(p, div);
```

- ❑ `hasChildNodes()` — возвращает значение `true`, если текущий узел имеет потомков, или `false` — в противном случае:

```
var p = document.getElementsByTagName('p')[0];
if (p.hasChildNodes()) console.log('Есть');
else console.log('Дочерних узлов нет');
```

Для получения информации об элементах таблицы предназначены специальные свойства объекта таблицы:

- ❑ `caption` — ссылка на тег `<caption>` (заголовок таблицы) или значение `null`, если он не существует;
- ❑ `tHead` — ссылка на тег `<thead>` (секцию шапки) или `null`, если он не существует;
- ❑ `tFoot` — ссылка на тег `<tfoot>` (секцию поддона) или `null`, если он не существует;
- ❑ `tBodies` — коллекция всех тегов `<tbody>` (секций основного содержимого) в таблице;
- ❑ `rows` — коллекция всех строк (тегов `<tr>`) в таблице, включая строки, содержащиеся в тегах `<thead>`, `<tbody>` и `<tfoot>`. Каждый элемент коллекции содержит следующие свойства:
- `cells` — коллекция всех ячеек в строке таблицы. Каждый элемент коллекции содержит свойство `cellIndex`, через которое доступен индекс ячейки в строке;
  - `rowIndex` — индекс строки в таблице;
  - `sectionRowIndex` — индекс строки внутри тега `<thead>`, `<tbody>` или `<tfoot>`.

Для создания и удаления элементов таблицы предназначены следующие методы:

- ❑ `createCaption()` — создает и возвращает новый тег `<caption>` или возвращает ссылку на существующий тег. Вызывается у объекта таблицы. Добавим заголовок к таблице:

```
var table = document.getElementsByTagName('table')[0];
var caption = table.createCaption();
var text = document.createTextNode('Это заголовок таблицы');
caption.appendChild(text); // Добавляем текст в тег <caption>
```

- ❑ `deleteCaption()` — удаляет тег `<caption>`. Если такого тега в таблице нет, ничего не делает. Вызывается у объекта таблицы;
- ❑ `createTHead()` — создает и возвращает новый тег `<thead>` или возвращает ссылку на существующий тег. Вызывается у объекта таблицы;
- ❑ `deleteTHead()` — удаляет тег `<thead>`. Если такого тега в таблице нет, ничего не делает. Вызывается у объекта таблицы;
- ❑ `createTBody()` — создает и возвращает новый тег `<tbody>`. Всегда создает новый тег, даже если в таблице уже имеется таковой. Вызывается у объекта таблицы;
- ❑ `createTFoot()` — создает и возвращает новый тег `<tfoot>` или возвращает ссылку на существующий тег. Вызывается у объекта таблицы;
- ❑ `deleteTFoot()` — удаляет тег `<tfoot>`. Если такого тега в таблице нет, ничего не делает. Вызывается у объекта таблицы;
- ❑ `insertRow([<Индекс>])` — создает новую строку по заданному индексу и возвращает ее в качестве результата. Если задать индекс `-1` или вообще опустить его, новая строка будет добавлена в конец таблицы (секции). Вызывается у объекта таблицы или секции;
- ❑ `deleteRow(<Индекс>)` — удаляет из таблицы (секции) строку по указанному индексу. Если указать индекс `-1`, будет удалена последняя строка. Вызывается у объекта таблицы или секции;
- ❑ `insertCell([<Индекс>])` — создает пустую ячейку по заданному индексу и возвращает ее в качестве результата. Если задать индекс `-1` или вообще опустить его, новая ячейка будет добавлена в конец строки. Вызывается у объекта строки. Добавим новую строку в самое начало первой таблицы на странице:

```
var table = document.getElementsByTagName('table')[0];
var row = table.insertRow(0);
var cell1 = row.insertCell(0);
var cell2 = row.insertCell(1);
var t1 = document.createTextNode('Первая ячейка');
var t2 = document.createTextNode('Вторая ячейка');
cell1.appendChild(t1);
cell2.appendChild(t2);
```

- ❑ `deleteCell(<Индекс>)` — удаляет из строки ячейку по указанному индексу. Если указать индекс `-1`, будет удалена последняя ячейка. Вызывается у объекта строки.

## 21.10. Общие свойства и методы элементов веб-страницы

Все элементы веб-страницы поддерживают следующие общие свойства:

- ❑ `id` — уникальный идентификатор текущего элемента, заданный параметром `id`;
- ❑ `className` — имена стилевых классов через пробел, заданные в параметре `class`;
- ❑ `classList` — коллекция стилевых классов из параметра `class`;
- ❑ `tagName` — имя тега элемента в нижнем регистре без символов `<` и `>`:

```
<div id="div1" class="cls1">Текст</div>
. . .
var elem = document.getElementById('div1');
console.log(elem.id); // div1
console.log(elem.className); // cls1
console.log(elem.tagName); // DIV
```

- ❑ `innerHTML` — HTML-код содержимого элемента:

```
var elem = document.getElementById('div1');
elem.innerHTML = 'Новый текст';
```

Результат в HTML-коде:

```
<div id="div1" class="cls1">Новый текст</div>
```

- ❑ `textContent` — текстовое содержимое элемента:

```
var elem = document.getElementById('div1');
elem.textContent = 'Новый текст';
console.log(elem.textContent); // Новый текст
```

- ❑ `attributes` — коллекция всех параметров тега текущего элемента. Каждый элемент коллекции содержит два свойства:

- `name` — имя параметра;
- `value` — значение параметра.

Пример:

```
var elem = document.getElementById('div1');
var arr = elem.attributes;
for (var i = 0; i < arr.length; i++) {
 console.log(arr[i].name + ' = ' + arr[i].value);
}
```

Общие методы элементов веб-страницы:

- ❑ `getAttribute(<Имя параметра>)` — возвращает значение параметра текущего элемента с заданным именем:

```
var elem = document.getElementById('div1');
console.log(elem.getAttribute('id')); // div1
```



- ❑ `setAttribute(<Имя параметра>, <Значение>)` — присваивает указанное значение параметру текущего элемента с заданным именем;
- ❑ `removeAttribute(<Имя параметра>)` — удаляет из текущего элемента параметр с заданным именем;
- ❑ `hasAttribute(<Имя параметра>)` — возвращает значение `true`, если в текущем элементе существует параметр с указанным именем, и `false` — в противном случае;
- ❑ `getElementsByTagName(<Имя тега>)` — возвращает ссылку на коллекцию дочерних элементов, созданных с использованием тега с заданным именем (указывается без символов `<` и `>`). Если подходящих элементов в текущем элементе нет, возвращает ссылку на пустую коллекцию;
- ❑ `getElementsByClassName(<Стилевой класс>)` — возвращает ссылку на коллекцию дочерних элементов, у которых задан указанный стиливой класс (указывается без точки в начале). Если подходящих элементов в текущем элементе нет, возвращает ссылку на пустую коллекцию;
- ❑ `querySelector(<CSS-селектор>)` — возвращает ссылку на первый дочерний элемент, соответствующий указанному CSS-селектору, или значение `null`, если подходящих элементов нет;
- ❑ `querySelectorAll(<CSS-селектор>)` — возвращает ссылку на коллекцию дочерних элементов, соответствующих указанному CSS-селектору. Если подходящих элементов нет, возвращает ссылку на пустую коллекцию;
- ❑ `scrollIntoView(true | false)` — прокручивает страницу так, чтобы текущий элемент оказался в поле зрения. Если параметр равен `true`, текущий элемент по возможности будет размещаться у верхнего края окна, а если `false` — то у нижнего края.

Мы можем обращаться к параметрам любого тега напрямую через одноименные свойства. Изменим интернет-адрес гиперссылки:

```
Текст ссылки
<script>
document.getElementById('link1').href = 'doc2.html';
</script>
```

Изменим интернет-адрес изображения:

```

<script>
document.getElementById('image1').src = 'image2.gif';
</script>
```

## 21.11. Работа с таблицами стилей

Свойство `style` хранит объект, который описывает стиль текущего элемента, указанный в параметре `style`. Свойства этого объекта одноименны соответствующим атрибутам стиля со следующими отличиями:

- ❑ символы – (дефис) удаляются;
- ❑ первые буквы всех слов (кроме первого) в имени атрибута делаются прописными.

Примеры преобразования атрибутов стиля в свойства объекта `style`:

```
color -> color
font-family -> fontFamily
font-size -> fontSize
border-left-style -> borderLeftStyle
```

Пример указания значений:

```
<div id="div1">Текст</div>
...
var elem = document.getElementById('div1');
elem.style.color = 'red';
elem.style.fontFamily = 'Verdana';
elem.style.fontSize = '14pt';
elem.style.borderLeftStyle = 'solid';
```

Чтобы получить результирующий стиль, составленный из всех стилей, которые действуют на текущий элемент, нужно использовать метод `getComputedStyle()` объекта `window`:

```
var elem = document.getElementById('div1');
console.log(window.getComputedStyle(elem, null).width); // 956px
```

Если выполняется попытка получения значения размера, которое было задано в одной из относительных единиц измерения, метод `getComputedStyle()` выдаст абсолютное значение, актуальное на текущий момент и представленное в пикселах.

Получить имена стилевых классов, указанных у текущего элемента страницы, позволяют следующие свойства:

- ❑ `className` — имя стилевого класса, заданное параметром `class`, или, если у элемента указано несколько стилевых классов, строка с их именами, перечисленными через пробелы;
- ❑ `classList` — коллекция всех стилевых классов из параметра `class`, представленная объектом класса `DOMTokenList`:

```
<div id="div1" class="cls1 cls2 cls3">Текст</div>
<script>
var elem = document.getElementById('div1');
console.log(elem.className); // cls1 cls2 cls3
console.log(elem.classList);
// DOMTokenList(3) ["cls1", "cls2", "cls3"]
</script>
```

Объект `DOMTokenList` содержит следующие свойства и методы:

- ❑ `length` — число элементов в коллекции;
- ❑ `add(<Класс 1>[, . . ., <Класс N>])` — добавляет заданные стилевые классы в коллекцию;
- ❑ `remove(<Класс 1>[, . . ., <Класс N>])` — удаляет указанные стилевые классы из коллекции;
- ❑ `item(<Индекс>)` — возвращает стиливый класс с указанным индексом.

Для извлечения стиливого класса также можно указать его индекс внутри квадратных скобок: `arr[<Индекс>]`;

- ❑ `toggle(<Класс>[, <Флаг>])` — если вторым параметром указано значение `undefined` или второй параметр опущен, удаляет заданный стиливый класс, если он существует, или добавляет заданный стиливый класс, если он отсутствует. Если во втором параметре указано значение `true`, добавляет заданный стиливый класс, а если `false` — удаляет его;
- ❑ `contains(<Класс>)` — возвращает `true`, если заданный стиливый класс присутствует в текущей коллекции, и `false` — в противном случае.

Пример приведен в листинге 21.1.

#### Листинг 21.1. Работа с таблицами стилей

```
<div id="div1" class="cls1 cls2">Текст</div>
<script>
var elem = document.getElementById('div1');
var arr = elem.classList;
for (let i = 0; i < arr.length; i++) {
 console.log(arr[i]);
 // console.log(arr.item(i));
}
arr.add('cls3');
console.log(arr); // DOMTokenList(3) ["cls1", "cls2", "cls3"]
arr.remove('cls3');
console.log(arr); // DOMTokenList ["cls1", "cls2"]
arr.toggle('cls3');
console.log(arr); // DOMTokenList(3) ["cls1", "cls2", "cls3"]
arr.toggle('cls3');
console.log(arr); // DOMTokenList ["cls1", "cls2"]
console.log(arr.contains('cls2')); // true
console.log(arr.contains('cls3')); // false
</script>
```

## 21.12. Объект *selection*: работа с выделением

Объект *selection* позволяет получить доступ к *выделению* — выделенному фрагменту страницы. Получить его можно вызовом метода `getSelection()` объекта `window`.

Объект *selection* содержит следующие свойства:

- `anchorNode` — ссылка на текстовый узел, в котором началось выделение, или значение `null`, если выделения нет. Получим элемент, в котором началось выделение, и сделаем его фон красным:

```
var rng = window.getSelection();
if (rng.anchorNode)
 rng.anchorNode.parentNode.style.backgroundColor = 'red';
```

- `anchorOffset` — смещение от начала текстового узла, возвращаемого свойством `anchorNode`, до начальной границы выделения;
- `focusNode` — ссылка на текстовый узел, в котором закончилось выделение, или значение `null`, если выделения нет;
- `focusOffset` — смещение от начала текстового узла, возвращаемого свойством `focusNode`, до конечной границы выделения:

```
var rng = window.getSelection();
if (rng.focusNode) {
 rng.focusNode.parentNode.style.backgroundColor = 'red';
 console.log(rng.focusOffset);
}
```

- `rangeCount` — число фрагментов, представленных объектами класса `Range`, которые входят в состав выделения;
- `isCollapsed` — возвращает `true`, если выделение свернуто в точку, и `false` — в противном случае.

Методы объекта *selection*:

- `toString()` — возвращает текстовое содержимое выделения:

```
var rng = window.getSelection();
if (!rng.isCollapsed) {
 console.log(rng.toString());
}
```

- `collapse()` и `setPosition()` — сворачивают выделение в точку, находящуюся в указанном узле по заданному смещению от его начала. Если смещение не задано, то выделение будет свернуто в начало указанного узла. Формат методов:

```
collapse | setPosition (<Узел>, [<Смещение>])
```

- `collapseToStart()` — сворачивает выделение в начало первого фрагмента, входящего в состав выделения;

- ❑ `collapseToEnd()` — сворачивает выделение в конец последнего фрагмента, входящего в состав выделения;
- ❑ `deleteFromDocument()` — удаляет выделенный фрагмент из страницы;
- ❑ `extend(<Узел>, [<Смещение>])` — перемещает конечную границу выделенного фрагмента в позицию, находящуюся в указанном узле по заданному смещению от его начала. Если смещение не задано, выделение будет свернуто в начало указанного узла. В качестве примера расширим (или уменьшим в зависимости от направления выделения) выделение до конца узла, содержащего конечную границу:

```
var rng = window.getSelection();
if (!rng.isCollapsed) {
 var len = rng.focusNode.length;
 if (rng.focusOffset != len) {
 rng.extend(rng.focusNode, len);
 }
}
else window.alert('Нет выделения');
```

- ❑ `containsNode(<Узел>[, <Флаг>])` — возвращает `true`, если в состав текущего выделения, полностью или частично, входит указанный узел, и `false` — в противном случае. Если вторым параметром задано значение `true`, метод вернет `true`, если указанный узел входит в состав выделения хотя бы частично. Если вторым параметром задано `false` или второй параметр опущен, метод вернет `true`, только когда указанный узел полностью входит в выделение;
- ❑ `getRangeAt(<Индекс>)` — возвращает фрагмент, входящий в состав текущего выделения и имеющий указанный индекс;
- ❑ `addRange(<Фрагмент>)` — добавляет указанный фрагмент к текущему выделению;
- ❑ `removeRange(<Фрагмент>)` — удаляет указанный фрагмент из текущего выделения;
- ❑ `empty()` и `removeAllRanges()` — удаляет все фрагменты из текущего выделения;
- ❑ `selectAllChildren(<Узел>)` — выделяет текстовое содержимое указанного узла и всех его потомков. Выделим содержимое первого абзаца на странице:

```
var rng = window.getSelection();
var elem = document.getElementsByTagName('p')[0];
rng.selectAllChildren(elem);
```

Текст, выделенный в текстовом поле или области редактирования, нельзя получить с помощью метода `getSelection()`. Вместо этого метода следует использовать свойства `selectionStart` (индекс символа, на котором находится начало выделения) и `selectionEnd` (индекс символа конца выделения) элемента формы. Пример:

```
<input type="text" name="txt1" id="txt1">
. . .
// Ссылка на текстовое поле
var elem = document.getElementById('txt1');
```

```
if (elem.selectionStart != undefined &&
 elem.selectionEnd != undefined) {
 var s = elem.selectionStart; // Начальная позиция
 var e = elem.selectionEnd; // Конечная позиция
 window.alert(elem.value.substring(s, e));
}
```

Выделить фрагмент внутри текстового поля (области редактирования) позволяет метод `setSelectionRange(<Начало>, <Конец>)`:

```
// Ссылка на текстовое поле
var elem = document.getElementById('txt1');
elem.focus();
elem.setSelectionRange(0, 5);
```

## 21.13. Объект *Range*: работа с фрагментами текста

Объект `Range` предоставляет произвольный фрагмент текста веб-страницы. Создать такой фрагмент позволяет метод `createRange()` объекта `document`:

```
var rng = document.createRange();
```

Также можно получить ссылку на фрагмент, входящий в состав выделения, с помощью метода `getRangeAt(<Индекс фрагмента>)` объекта `selection`:

```
var sel = window.getSelection();
if (!sel.isCollapsed) {
 var rng = sel.getRangeAt(0);
}
```

Свойства объекта `Range`:

- `startContainer` — ссылка на узел, в котором содержится начальная точка фрагмента;
- `startOffset` — смещение от начала узла, возвращаемого свойством `startContainer`, до начальной точки фрагмента;
- `endContainer` — ссылка на узел, в котором содержится конечная точка фрагмента;
- `endOffset` — смещение от начала узла, возвращаемого свойством `endContainer`, до конечной точки фрагмента.

Пример:

```
<p id="txt1">Текст внутри абзаца</p>
. . .
var elem = document.getElementById('txt1').firstChild;
var rng = document.createRange();
rng.setStart(elem, 1); // Начальная точка
rng.setEnd(elem, 5); // Конечная точка
```

```
console.log(rng.startContainer);
console.log(rng.startOffset);
console.log(rng.endContainer);
console.log(rng.endOffset);
var sel = window.getSelection();
sel.addRange(rng); // Выделяем фрагмент
```

- ❑ `collapsed` — значение `true`, если фрагмент свернут в точку, и `false` — в противном случае;
- ❑ `commonAncestorContainer` — ссылка на узел, в котором содержатся начальная и конечная точки фрагмента (обычно это родитель узлов, содержащих начальную и конечную точки).

Методы объекта `Range`:

- ❑ `toString()` — возвращает текстовое содержимое фрагмента;
- ❑ `setStart(<Узел>, <Смещение>)` — устанавливает положение начальной точки фрагмента;
- ❑ `setEnd(<Узел>, <Смещение>)` — устанавливает положение конечной точки фрагмента;
- ❑ `setStartBefore(<Узел>)` — устанавливает начальную точку фрагмента перед указанным узлом;
- ❑ `setStartAfter(<Узел>)` — устанавливает начальную точку фрагмента после указанного узла;
- ❑ `setEndBefore(<Узел>)` — устанавливает конечную точку фрагмента перед указанным узлом;
- ❑ `setEndAfter(<Узел>)` — устанавливает конечную точку фрагмента после указанного узла;
- ❑ `cloneRange()` — создает и возвращает копию текущего фрагмента;
- ❑ `cloneContents()` — создает и возвращает копию внутреннего содержимого текущего фрагмента в виде объекта `DocumentFragment`;
- ❑ `deleteContents()` — удаляет со страницы все внутреннее содержимое фрагмента;
- ❑ `extractContents()` — удаляет со страницы все внутреннее содержимое фрагмента и возвращает объект `DocumentFragment`, в котором находится удаленное содержимое;
- ❑ `collapse(<true | false>)` — сворачивает фрагмент в точку. Если в качестве параметра указано значение `true`, фрагмент сворачивается в начальную точку, а если `false` — в конечную точку;
- ❑ `selectNode(<Узел>)` — ограничивает фрагмент указанным в качестве параметра узлом;
- ❑ `selectNodeContents(<Узел>)` — ограничивает фрагмент внутренним содержимым указанного узла;

- ❑ `insertNode(<Узел>)` — вставляет новый узел в начало текущего фрагмента;
- ❑ `surroundContents(<Узел>)` — вкладывает содержимое текущего фрагмента в указанный узел;
- ❑ `compareBoundaryPoints(<Точки сравнения>, <Фрагмент>)` — сравнивает позиции текущего и заданного во втором параметре фрагментов. В качестве первого параметра могут быть указаны следующие значения:
  - 0 — `START_TO_START` — сравнение начальных точек;
  - 1 — `START_TO_END` — сравнение начальной точки указанного фрагмента с конечной точкой текущего;
  - 2 — `END_TO_END` — сравнение конечных точек;
  - 3 — `END_TO_START` — сравнение конечной точки заданного фрагмента с начальной точкой текущего.

В качестве примера использования объекта `Range` найдем внутри абзаца слово "фрагмент" и вложим его в тег `<strong>` (листинг 21.2).

**Листинг 21.2. Поиск заданного слова и заключение его в HTML-тег**

```
<p id="txt">Текст для выделения фрагмента</p>
<input type="button" id="btn1" onclick="handler()" value="Выделить">
<script>
function handler() {
 if (document.createRange) {
 var p = document.getElementById('txt').firstChild;
 var text = p.nodeValue; // Получаем текст абзаца
 var ind = text.indexOf('фрагмент');
 if (ind != -1) { // Если текст найден
 // Создаем объект Range
 var rng = document.createRange();
 rng.setStart(p, ind); // Начальная точка
 // Конечная точка
 rng.setEnd(p, ind + 8);
 // Элемент, в который будем вкладывать текст
 var s = document.createElement('strong');
 // Вкладываем фрагмент в тег strong
 rng.surroundContents(s);
 }
 }
 else {
 window.alert('Веб-браузер не поддерживает метод createRange()');
 }
}
</script>
```



Изменим цвет фона фрагмента, выделенного пользователем (листинг 21.3).

### Листинг 21.3. Изменение цвета фона выделенного фрагмента

```
<p id="txt">Текст для выделения фрагмента</p>
<input type="button" id="btn1" onclick="handler()" value="Выделить">
<script>
function handler() {
 if (document.createRange && window.getSelection()) {
 var sel = window.getSelection();
 if (!sel.isCollapsed) {
 var rng = sel.getRangeAt(0);
 sel.collapseToStart(); // Убираем выделение
 // Элемент, в который будем вкладывать выделенный текст
 var s = document.createElement('span');
 s.style.backgroundColor = '#FFE9B3';
 // Вкладываем фрагмент в тег span
 rng.surroundContents(s);
 }
 else window.alert('Нет выделенного фрагмента');
 }
 else {
 window.alert('Веб-браузер не поддерживает метод createRange()');
 }
}
</script>
```

## 21.14. Cookie: хранение данных на компьютере клиента

Веб-браузеры позволяют сохранить произвольное значение под указанным именем в cookie — текстовом файле, размещаемом на компьютере пользователя. Число сохраняемых значений (и соответственно создаваемых cookie) не ограничено.

Возможность применения cookie можно отключить в настройках веб-браузера. Проверить возможность использования cookie позволяет свойство `cookieEnabled` объекта `navigator`:

```
if (navigator.cookieEnabled) {
 window.alert('Использование cookie разрешено');
}
```

Запись cookie осуществляется путем присвоения свойству `cookie` объекта `document` строкового значения в следующем формате:

```
document.cookie = '<Имя>=<Значение>; [expires=<Дата>;]
 [domain=<Имя домена>;] [path=<Путь>;] [secure;]';
```

Здесь задаются следующие параметры:

- `<Имя>=<Значение>` — имя сохраняемого значения и само значение (единственный обязательный параметр);
- `expires` — дата удаления cookies в следующем формате:

Thu, 01 Jan 1970 00:00:01 GMT

Получить дату в этом формате можно с помощью методов `setTime()` и `toUTCString()` класса `Date`. Методу `setTime()` нужно передать текущее время плюс время хранения cookie в миллисекундах. Текущее время можно получить с помощью метода `getTime()`. Рассчитать время хранения cookies можно исходя из следующих соотношений:

- 1 секунда = 1000 миллисекунд;
- 1 минута = 60 секунд = 60 000 миллисекунд;
- 1 час = 60 минут = 3600 секунд = 3 600 000 миллисекунд;
- 1 день = 24 часа = (24×3 600 000) миллисекунд = 86 400 000 миллисекунд.

Например:

```
var d = new Date();
d.setTime(d.getTime() + 3600000); // Задан 1 час
var endDate = d.toUTCString(); // Дата удаления cookie
```

Если параметр `expires` не задан, то по истечении текущего сеанса работы веб-браузера cookie будет автоматически удален;

- `domain=<Имя домена>` — доменная часть интернет-адреса, с которой связывается данный cookie;
- `path=<Путь>` — путь, присутствующий в интернет-адресе, с которым связывается данный cookie.

Считывание cookie выполняется также с помощью свойства `cookie` объекта `document`:

```
var cke = document.cookie;
```

Переменная `cke` будет содержать строку, в которой перечислены все установленные пары `имя=значение` через точку с запятой:

```
"имя1=значение1; имя2=значение2"
```

Для удаления cookie следует записать cookie с произвольным значением и прошедшей датой.

В качестве примера сохраним имя и фамилию пользователя и при следующем посещении будем приветствовать его, используя сохраненные данные (листинг 21.4). Добавим также возможность удаления cookie. Для совместимости закодируем введенные данные с помощью метода `encodeURIComponent()`, а при выводе раскодировем их с помощью метода `decodeURIComponent()`. Это позволяет безопасно сохранять значения, введенные кириллицей.

**Листинг 21.4. Установка и удаление cookie**

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>Установка и удаление cookie</title>
</script>
function saveCookies() {
 if (navigator.cookieEnabled) {
 var txt1 = document.getElementById('txt1');
 var txt2 = document.getElementById('txt2');
 if (txt1.value != '' && txt2.value != '') {
 var d = new Date();
 d.setTime(d.getTime() + 3600000); // Задан 1 час
 var endDate = d.toUTCString(); // Дата удаления cookie
 var str = 'name1=' + encodeURIComponent(txt1.value);
 str += ';expires=' + endDate + ';';
 document.cookie = str;
 str = 'name2=' + encodeURIComponent(txt2.value);
 str += ';expires=' + endDate + ';';
 document.cookie = str;
 txt1.value = '';
 txt2.value = '';
 showCookies();
 }
 else {
 window.alert('Не заполнено обязательное поле');
 }
 }
}

function deleteCookies() {
 if (navigator.cookieEnabled) {
 if (document.cookie != '') {
 var d = new Date();
 d.setTime(1000); // Дата в прошлом
 var endDate = d.toUTCString();
 document.cookie = 'name1=;expires=' + endDate + ';';
 document.cookie = 'name2=;expires=' + endDate + ';';
 showCookies();
 }
 }
}

function showCookies() {
 if (navigator.cookieEnabled) {
 var div1 = document.getElementById('div1');
```

```
if (document.cookie != '') {
 var arr1 = [], arr2 = [];
 var obj = {};
 var str = document.cookie;
 if (str.indexOf('; ') != -1) {
 arr1 = str.split('; ');
 for (var i = 0, c = arr1.length; i < c; i++) {
 arr2 = arr1[i].split('=');
 obj[arr2[0]] = arr2[1];
 }
 }
 else {
 arr2 = str.split('=');
 obj[arr2[0]] = arr2[1];
 }
 var msg = 'Привет, ';
 msg += decodeURIComponent(obj.name2).replace('<', '<');
 msg += ' ' + decodeURIComponent(obj.name1).replace('<', '<');
 div1.innerHTML = msg;
}
else div1.innerHTML = '';
}
}

window.onload = function() {
 showCookies();
}
</script>
</head>
<body>
<div id="div1"></div>
<div>
Введите ваше имя:

<input type="text" id="txt1">

Введите вашу фамилию:

<input type="text" id="txt2">

<input type="button" value="Сохранить" onclick="saveCookies()">

<input type="button" value="Удалить cookie" onclick="deleteCookies()">
</div>
</body>
</html>
```

## 21.15. Хранилище

Мы уже знаем способ сохранить произвольные данные на стороне клиента — cookie. Однако для этой цели удобнее пользоваться встроенным *хранилищем*, позволяющим сохранять на стороне клиента произвольное число значений любого типа и любого объема.

### 21.15.1. Сессионное и локальное хранилища

Современные веб-браузеры поддерживают два различных хранилища:

- *Сессионное хранилище* — хранит данные, пока в окне веб-браузера открыта текущая страница. После перехода на другую страницу или закрытия окна все сохраненные данные будут удалены. Применяется для временного сохранения данных на случай случайного обновления страницы посетителем.
- *Локальное хранилище* — хранит данные в течение произвольного времени, даже после перехода на другую страницу или закрытия окна веб-браузера.

Хранилище любого типа хранит данные, относящиеся к какой-то одной странице. Данные, сохраненные любой другой страницей, при этом считать невозможно — так сделано ради безопасности.

### 21.15.2. Работа с хранилищем

Сессионное хранилище доступно через свойство `sessionStorage`, а локальное — через свойство `localStorage` объекта `window`.

#### **ВНИМАНИЕ!**

Хранилища обоих типов доступны лишь в том случае, если страница была загружена с веб-сервера. При загрузке страницы с локального диска свойства `sessionStorage` и `localStorage` могут быть недоступны.

Хранилище любого типа представляется объектом класса `Storage`. Все манипуляции по сохранению и чтению данных выполняются посредством вызова методов этого объекта. Пример:

```
var stSession = window.sessionStorage;
// Получаем сессионное хранилище.
if (window.localStorage) {
 var stLocal = window.localStorage;
 console.log(stLocal);
 // Работаем с локальным хранилищем
}
else {
 console.log('Веб-браузер не поддерживает локальное хранилище');
```

Метод `setItem(<Имя>, <Значение>)` сохраняет в хранилище указанное значение под заданным именем:

```
stLocal.setItem('language', 'JavaScript');
```

Метод `getItem(<Имя>)` считывает из хранилища и возвращает значение с указанным именем. Если значение с таким именем отсутствует в хранилище, возвращается `null`. Пример:

```
var language = stLocal.getItem('language');
if (language) {
```

```
// Используем полученное из хранилища значение
console.log(language);
}
else {
 console.log('Такого значения в хранилище нет');
}
```

Метод `removeItem(<Имя>)` удаляет из хранилища значение с заданным именем:

```
stLocal.removeItem('language');
```

Единственное свойство объекта `Storage` — `length` — содержит число сохраненных в хранилище значений в виде числа.

#### **ПРИМЕЧАНИЕ**

Посмотреть содержимое хранилища можно на вкладке **Хранилище** Инструментов разработчика. Быстро отобразить эту вкладку можно нажатием комбинации клавиш `<Shift>+<F9>`.

### **21.15.3. Использование локального хранилища для хранения данных**

Локальное хранилище идеально подходит для временного хранения данных, введенных посетителем в форму. Код из листинга 21.5 демонстрирует его использование для этой цели.

**Листинг 21.5. Пример временного хранения данных в хранилище**

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>Использование хранилища</title>
<script>
 window.addEventListener('load', function() {
 var stLocal = window.localStorage;
 var txtTitle = document.getElementById('txtTitle');
 var txtContent = document.getElementById('txtContent');
 var btnSave = document.getElementById('btnSave');
 var btnLoad = document.getElementById('btnLoad');
 btnLoad.disabled = !((stLocal.getItem('title')) &&
 (stLocal.getItem('content')));
 btnSave.addEventListener('click', function() {
 if ((txtTitle.value) && (txtContent.value)) {
 stLocal.setItem('title', txtTitle.value);
 stLocal.setItem('content', txtContent.value);
 btnLoad.disabled = false;
 }
 }, false);
```

```

 btnLoad.addEventListener('click', function() {
 txtTitle.value = stLocal.getItem('title');
 txtContent.value = stLocal.getItem('content');
 }, false);
 }, false);
</script>
</head>
<body>
 <form>
 <p>Заголовок статьи

 <input type="text" id="txtTitle"></p>
 <p>Содержимое статьи

 <textarea id="txtContent"></textarea></p>
 <p><input type="submit" value="Отправить"></p>
 <p><input type="button" id="btnSave" value="Сохранить">
 <input type="button" id="btnLoad" value="Загрузить" disabled></p>
 </form>
</body>
</html>

```

## 21.16. Работа с графическими изображениями

Объект, представляющий графическое изображение, поддерживает следующие дополнительные свойства:

- `complete` — значение `true`, если изображение было полностью загружено, и `false` — в противном случае;
- `naturalWidth` — ширина загруженного изображения в пикселах. Отличается от значения параметра `width`, который задает отображаемую ширину изображения;
- `naturalHeight` — высота загруженного изображения в пикселах. Отличается от значения параметра `height`, который задает отображаемую высоту изображения.

Пример приведен в листинге 21.6.

### Листинг 21.6. Получение сведений о графическом изображении

```


<script>
window.onload = function() {
 var img = document.getElementById('img');
 console.log(img.complete);
 console.log(img.naturalWidth);
 console.log(img.naturalHeight);
 console.log(img.width);
 console.log(img.height);
}
</script>

```

## 21.17. Работа с мультимедиа

Аудио- и видеопроигрыватели HTML поддерживают расширенный набор программных инструментов.

Объекты аудио- и видеопроигрывателей поддерживают следующие дополнительные свойства:

- ☐ `src` — интернет-адрес файла с роликом, записанный в параметре `src` тега `<audio>` или `<video>`;
- ☐ `currentSrc` — интернет-адрес воспроизводящегося в данный момент ролика. Может отличаться от интернет-адреса, заданного в параметре `src`, если ролик предоставляется серверным приложением, автоматически выбирающим мультимедийный файл в зависимости от параметров клиентского компьютера;
- ☐ `poster` — интернет-адрес файла заставки для видео;
- ☐ `preload` — признак, следует ли выполнять предварительную загрузку ролика. Значением этого свойства должна быть одна из строк:
  - `'none'` — не выполнять предзагрузку;
  - `'metadata'` — загрузить только заголовок файла, где хранятся сведения о ролике;
  - `'auto'` — загрузить весь файл; значение по умолчанию;
- ☐ `width` — отображаемая ширина видеоролика в пикселах;
- ☐ `height` — отображаемая высота видеоролика в пикселах;
- ☐ `videoWidth` — изначальная ширина видеоролика в пикселах;
- ☐ `videoHeight` — изначальная высота видеоролика в пикселах;
- ☐ `autoplay` — если `true`, сразу после загрузки ролика он будет запущен на воспроизведение, если `false` — не будет запущен;
- ☐ `controls` — если `true`, на экране будут присутствовать элементы для управления воспроизведением ролика, если `false` — не будут присутствовать;
- ☐ `loop` — если `true`, ролик воспроизводится бесконечно, если `false` — всего один раз;
- ☐ `volume` — громкость звука в виде числа с плавающей точкой от 0.0 (звук отсутствует) до 1.0 (максимальная громкость);
- ☐ `muted` — если `true`, звук будет приглушен, если `false` — не будет приглушен;
- ☐ `duration` — продолжительность ролика в виде числа в секундах;
- ☐ `currentTime` — текущая позиция воспроизведения ролика в виде числа в секундах;
- ☐ `paused` — если `true`, воспроизведение ролика будет приостановлено, если `false` — возобновлено;



- ❑ `ended` — значение `true`, если воспроизведение ролика закончилось, и `false` — в противном случае;
- ❑ `readyState` — состояние загрузки ролика в виде числа:
  - 0 — загрузка еще не началась;
  - 1 — загружен лишь заголовок файла, хранящий сведения о самом ролике;
  - 2 — загружены данные, достаточные для вывода текущего кадра, но не для дальнейшего воспроизведения;
  - 3 — загружены данные, достаточные для вывода текущего кадра и, по крайней мере, для начала воспроизведения;
  - 4 — загружены данные, достаточные для бесперебойного воспроизведения всего ролика;
- ❑ `networkState` — текущее состояние загрузки ролика в виде числа:
  - 0 — загрузка еще не началась;
  - 1 — ролик уже загружен;
  - 2 — идет загрузка ролика;
  - 3 — файл с роликом отсутствует на сервере;
- ❑ `playbackRate` — текущая скорость воспроизведения ролика. Значение должно представлять собой число с плавающей точкой, которое будет умножено на величину скорости воспроизведения, полученную из файла с роликом; положительные значения задают воспроизведение в прямом порядке, отрицательные — в обратном;
- ❑ `seeking` — значение `true`, если в данный момент посетитель меняет позицию воспроизведения ролика, и `false` — в противном случае.

Аудио- и видеопроигрыватели поддерживают следующие методы:

- ❑ `load()` — выполняет повторную загрузку ролика;
- ❑ `pause()` — приостанавливает воспроизведение ролика;
- ❑ `play()` — запускает или возобновляет воспроизведение ролика.
- ❑ `canPlayType(<MIME-тип>)` — позволяет узнать, поддерживает ли веб-браузер воспроизведение роликов формата, соответствующего указанному в виде строки MIME-типу. Возвращает одно из следующих строковых значений:
  - `'probably'` — скорее всего, поддерживает;
  - `'maybe'` — возможно, поддерживает;
  - `''` (пустая строка) — гарантированно не поддерживает.

## 21.18. Средства геолокации

Современные веб-браузеры предоставляют средства для *геолокации* — получения географических координат клиентского компьютера.

### 21.18.1. Доступ к средствам геолокации

Средства геолокации доступны из свойства `geolocation` объекта `navigator`. Они представляют собой объект класса `Geolocation`, с помощью методов которого и выполняется получение местоположения. Пример:

```
if (navigator.geolocation) {
 var oGeo = navigator.geolocation;
 // Получаем данные геолокации
} else {
 // Веб-браузер не поддерживает геолокацию
}
```

### 21.18.2. Получение данных геолокации

Получить координаты клиентского компьютера однократно можно, обратившись к методу `getCurrentPosition()` класса `Geolocation`:

```
getCurrentPosition(<Функция для получения данных>[,
 <Функция, вызываемая в случае ошибки>[, <Параметры>]])
```

Единственным обязательным параметром этому методу передается функция, которая будет вызвана при успешном получении данных геолокации. В качестве единственного параметра она получит объект класса `GeolocationPosition`, хранящий данные геолокации.

У объекта класса `GeolocationPosition`, полученного функцией, нас интересует свойство `coords`. Оно хранит объект класса `GeolocationCoordinates`, представляющий координаты клиента и поддерживающий следующие свойства:

- ❑ `latitude` — широта в градусах;
- ❑ `longitude` — долгота в градусах;
- ❑ `altitude` — высота над уровнем моря в метрах или `null`, если значение высоты получить не удалось;
- ❑ `accuracy` — точность определения широты и долготы в метрах;
- ❑ `altitudeAccuracy` — точность определения высоты в метрах или `null`, если значение высоты получить не удалось;
- ❑ `heading` — направление движения в виде угла в градусах, отмеренного от направления на север по часовой стрелке, `NaN`, если клиентский компьютер никуда не движется, или `null`, если значение направления получить не удалось;
- ❑ `speed` — скорость движения в метрах в секунду, `NaN`, если клиентский компьютер никуда не движется, или `null`, если значение скорости получить не удалось.

Все эти свойства возвращают числовые значения.

Пример приведен в листинге 21.7.

#### Листинг 21.7. Получение данных геолокации

```
if (navigator.geolocation) {
 navigator.geolocation.getCurrentPosition(function(pos) {
 var lat = pos.coords.latitude;
 var lon = pos.coords.longitude;
 var alt = pos.coords.altitude;
 var heading = pos.coords.heading;
 if (isNaN(heading)) heading = 0;
 var speed = pos.coords.speed;
 });
}
```

### 21.18.3. Обработка нештатных ситуаций

При получении данных геолокации возможны нештатные ситуации (например, ошибки в работе инструментов определения местоположения). При их возникновении будет выполнена функция, переданная методу `getCurrentPosition()` вторым параметром.

В качестве единственного параметра эта функция получит объект класса `GeolocationPositionError`, хранящий сведения о возникшей ошибке. Этот класс поддерживает два свойства:

❑ `code` — числовой код ошибки:

- 1 — посетитель запретил странице доступ к данным геолокации;
- 2 — в работе инструментов для определения местоположения, встроенных в компьютер, возникла ошибка;
- 3 — истекло время, отведенное на определение местоположения;

❑ `message` — текстовое описание ошибки.

Пример приведен в листинге 21.8.

#### Листинг 21.8. Обработка нештатных ситуаций

```
navigator.geolocation.getCurrentPosition(function(pos) {
 // Получаем данные геолокации
}, function(error) {
 switch (error.code) {
 case 1:
 window.alert('Нет доступа к данным геолокации');
 break;
```

```
case 2:
 window.alert('Внутренняя ошибка');
 break;
case 3:
 window.alert('Истекло время, отведенное на получение данных');
 break;
}
});
```

## 21.18.4. Задание дополнительных параметров

При получении сведений о местоположении можно задать дополнительные параметры. Они указываются третьим параметром метода `getCurrentPosition()` в виде объекта класса `PositionOption`, поддерживающего следующие свойства:

- ❑ `enableHighAccuracy` — если `true`, будут предоставляться по возможности максимально точные данные геолокации, если `false` — данные меньшей точности. Получение точных данных может отнять больше времени и повысить энергопотребление устройства. Значение по умолчанию: `false`;
- ❑ `maximumAge` — время, в течение которого полученные данные геолокации будут кэшироваться устройством, в виде числа в миллисекундах. Если задано `0`, устройство не будет кэшировать данные, если `Infinity` — данные будут кэшироваться до тех пор, пока компьютер не изменит свое местоположение. Значение по умолчанию: `Infinity`;
- ❑ `timeout` — время, отведенное на получение сведений о местоположении, в виде числа в миллисекундах. По истечении этого времени будет сгенерирована ошибка. Если задано `Infinity`, то время ограничиваться не будет. Значение по умолчанию: `Infinity`.

В качестве примера зададим получение точных координат, кэширование их в течение 10 секунд и время на их получение, равное 5 секундам (листинг 21.9).

### Листинг 21.9. Пример указания дополнительных параметров метода `getCurrentPosition()`

```
navigator.geolocation.getCurrentPosition(function(pos) {
 // Получаем данные геолокации
}, function(error) {
 // Отслеживаем нештатные ситуации
}, {
 enableHighAccuracy: true,
 maximumAge: 10000,
 timeout: 5000
});
```

## 21.18.5. Отслеживание местоположения компьютера

Наконец, можно постоянно отслеживать местоположение клиентского компьютера, чтобы выяснить, куда он перемещается и достиг ли точки назначения. Для этого нужно использовать метод `watchPosition()`, формат вызова которого совпадает с таковым у метода `getCurrentPosition()`.

Как только местоположение клиентского компьютера изменится, будет вызвана функция, переданная этому методу первым параметром. А при возникновении нестандартной ситуации — функция, переданная вторым параметром.

Метод `watchPosition()` возвращает особый идентификатор, который позволяет в дальнейшем отменить отслеживание местоположения. Это можно сделать вызовом не возвращающего результат метода `clearWatch()`:

```
<Объект геолокации>.clearWatch(<Идентификатор>)
```

В листинге 21.10 приведен код, отслеживающий местоположение клиента, чтобы выяснить, достиг ли он точки назначения. Координаты точки назначения хранятся в переменных `latD` (широта) и `lonD` (долгота).

### Листинг 21.10. Пример отслеживания местоположения компьютера

```
var wp = navigator.geolocation.watchPosition(function(pos) {
 if ((latD == pos.coords.latitude) && (lonD == pos.coords.longitude)) {
 navigator.geolocation.clearWatch(wp);
 window.alert('Вы в точке назначения!');
 }
}, function(error) {
 window.alert(error.message);
});
```



## ГЛАВА 22

# События

Сама веб-страница и ее элементы регулярно генерируют разнообразные события: при щелчке мышью, при наведении курсора мыши, при нажатии клавиши, при окончании загрузки страницы или графического изображения и др. Чтобы отреагировать на эти события, достаточно назначить им соответствующие обработчики (подробнее о событиях и обработчиках рассказывалось в *разд. 14.6*).

### 22.1. Назначение обработчиков событий

Назначить обработчик события, генерируемого каким-либо элементом или самой страницей, можно несколькими способами.

Первый способ заключается в добавлении к тегу, создающему нужный элемент страницы, параметра, имя которого совпадает с именем обрабатываемого события. В качестве значения такого параметра указывается непосредственно программный код обработчика события (обычно — вызов функции, которая и обработает событие). Пример назначения функции `handler1()` в качестве обработчика нажатия кнопки:

```
<script>
 function handler1() {
 console.log('Нажата кнопка 1');
 }
</script>
<input type="button" value="Кнопка 1" onclick="handler1()">
```

Второй способ — присваивание ссылки на функцию-обработчик свойству элемента страницы, имя которого совпадает с обрабатываемым событием. Функция-обработчик может быть как именованной, так и анонимной. Пример:

```
function handler2() {
 console.log('Нажата кнопка 2');
}
document.getElementById('btn2').onclick = handler2;
```

```
document.getElementById('btn3').onclick = function() {
 console.log('Нажата кнопка 3');
};
```

Не забывайте, что при присваивании ссылки на функцию название функции указывается без круглых скобок. Если указать круглые скобки, функция будет вызвана, и свойству будет присвоен возвращенный ею результат.

Первые два способа позволяют назначить только один обработчик. Попытка присвоить свойству другой обработчик приведет к удалению ранее назначенного.

Третий способ позволяет назначить событию произвольное число обработчиков, которые будут выполняться последовательно. Для назначения обработчиков применяется метод `addEventListener()`. Формат метода:

```
addEventListener(<Событие>, <Ссылка на функцию-обработчик>[, <Фаза>]);
```

В параметре `<Событие>` указывается имя события в виде строки без префикса `on` (например, `'click'` вместо `onclick`). Описание параметра `<Фаза>` см. в *разд. 22.7*.

Указанной функции-обработчику в качестве параметра будет передана ссылка на объект класса `Event`, хранящий сведения о событии и позволяющий управлять им. Внутри функции через указатель `this` доступна ссылка на элемент, которому был назначен текущий обработчик.

Пример назначения обработчиков различными способами приведен в листинге 22.1.

#### Листинг 22.1. Назначение обработчиков событий

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>Назначение обработчиков событий</title>
</head>
<body>
 <input type="button" value="Кнопка 1" onclick="handler1()">
 <input type="button" value="Кнопка 2" id="btn2">
 <input type="button" value="Кнопка 3" id="btn3">
 <input type="button" value="Кнопка 4" id="btn4">
 <script>
function handler1() {
 console.log('Нажата кнопка 1');
}
function handler2() {
 console.log('Нажата кнопка 3');
}
function handler3(e) {
 console.log('Нажата кнопка 4. handler3()');
}
```

```
function handler4(e) {
 console.log('Нажата кнопка 4. handler4()');
}
document.getElementById('btn2').onclick = function() {
 console.log('Нажата кнопка 2');
};
// Имя функции указывается без круглых скобок
document.getElementById('btn3').onclick = handler2;
// Можно назначить сразу несколько обработчиков
var btn4 = document.getElementById('btn4');
btn4.addEventListener('click', handler3);
btn4.addEventListener('click', handler4);
</script>
</body>
</html>
```

## 22.2. Удаление обработчиков событий

Если обработчик события назначался через параметр тега или свойство, то для его удаления достаточно присвоить этому же свойству значение `null`, пустую строку или пустую анонимную функцию (последний способ работает во всех веб-браузерах):

```
document.getElementById('btn2').onclick = function() {};
```

Если обработчик назначался вызовом метода `addEventListener()`, удалить его можно вызовом метода `removeEventListener()`. Формат метода:

```
removeEventListener(<Событие>, <Ссылка на функцию-обработчик>[, <Фаза>]);
```

Пример:

```
document.getElementById('btn4').removeEventListener('click', handler3);
```

### **ОБРАТИТЕ ВНИМАНИЕ**

Нужно обязательно иметь ссылку на функцию-обработчик, ранее назначенную с помощью метода `addEventListener()`. В противном случае удалить обработчик не получится.

## 22.3. Указатель *this*

При назначении обработчика события с помощью свойства или метода `addEventListener()` внутри обработчика будет доступна ссылка на элемент, которому был назначен обработчик, через указатель `this`. Если обработчик назначается через параметр тега, указатель нужно передать в качестве параметра. Получим текст на кнопке при ее нажатии (листинг 22.2).



**Листинг 22.2. Указатель this**

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>Указатель this</title>
</head>
<body>
<input type="button" value="Кнопка 1" onclick="handler1(this)">
<input type="button" value="Кнопка 2" id="btn2">
<input type="button" value="Кнопка 3" id="btn3">
<script>
function handler1(elem) {
 console.log('Нажата кнопка ' + elem.value);
}
function handler2(event) {
 console.log('Нажата кнопка ' + this.value);
}
document.getElementById('btn2').onclick = function(event) {
 console.log('Нажата кнопка ' + this.value);
};
var btn3 = document.getElementById('btn3');
btn3.addEventListener('click', handler2);
</script>
</body>
</html>
```

## 22.4. Объект *event*

Объект класса `Event`, передаваемый обработчику через единственный параметр, позволяет получить детальную информацию о произошедшем событии и в некоторой мере управлять им. Если обработчик назначается через параметр тега, то объект события нужно передать в обработчик явно.

Объект `event` имеет следующие основные свойства:

- ❑ `type` — тип события в виде строки. Возвращается в нижнем регистре и без префикса `on`. Например, при событии `onclick` свойство `type` будет хранить строку `'click'`;
- ❑ `target` — ссылка на элемент, в котором возникло событие;
- ❑ `currentTarget` — возвращает ссылку на элемент, которому был назначен текущий обработчик события. Ссылается на тот же элемент, что и ключевое слово `this` внутри обработчика события. Значение свойства `currentTarget` может не совпадать со значением свойства `target`;
- ❑ `timeStamp` — время возникновения события в миллисекундах.

Можно назначить один обработчик сразу для нескольких элементов и даже для нескольких типов событий. В листинге 22.3 приведен пример обработки нажатия нескольких кнопок внутри одного обработчика.

#### Листинг 22.3. Объект event

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>Объект event</title>
</head>
<body>
<input type="button" value="Кнопка 1" onclick="handler(event)">
<input type="button" value="Кнопка 2" id="btn2">
<input type="button" value="Кнопка 3" id="btn3">
<script>
function handler(event) {
 console.log(event.type);
 console.log(event.target.value);
 console.log(event.currentTarget.value);
 console.log(event.timeStamp);
}
document.getElementById('btn2').onclick = handler;
var btn3 = document.getElementById('btn3');
btn3.addEventListener('click', handler);
</script>
</body>
</html>
```

## 22.5. Действия по умолчанию и их отмена

Для многих событий назначены *действия по умолчанию*, которые веб-браузер выполняет самостоятельно. Например, при щелчке на гиперссылке действием по умолчанию будет переход по указанному в гиперссылке интернет-адресу, нажатие кнопки **Отправить** приводит к отправке данных из формы и т. д.

Иногда действия по умолчанию необходимо прервать. Например, при отправке данных формы можно проверить их на соответствие заданным правилам и, если они не соответствуют, прервать отправку. Для этого используются следующие свойства и методы объекта event:

- ❑ `cancelable` — содержит `true`, если действие по умолчанию может быть отменено, и `false` — в противном случае;
- ❑ `preventDefault()` — отменяет действие по умолчанию, если его можно отменить;

- `defaultPrevented` — содержит `true`, если действие по умолчанию для текущего события было отменено в этом или предыдущем обработчике, и `false` — в противном случае.

Для отмены действия по умолчанию также можно вернуть из обработчика значение `false`.

В листинге 22.4 приведен пример проверки правильности ввода адреса электронной почты, а также два способа прерывания действия по умолчанию — перехода по гиперссылке: возвратом значения `false` и с помощью свойств и методов объекта `event`.

#### Листинг 22.4. Прерывание действий по умолчанию

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>Прерывание действий по умолчанию</title>
<script>
function testEmail() {
 var p = /^[a-z0-9_.-]+@[a-z0-9-]+\.[a-z]{2,6}$/i;
 // Получаем значение поля email
 var email = document.getElementById('email').value;
 if (p.test(email)) {
 if (window.confirm('Отправить данные формы?')) {
 return true; // Отправляем
 }
 else return false; // Прерываем
 }
 else {
 alert('E-mail введен неправильно');
 return false; // Прерываем
 }
}

function handler(e) {
 e = e || window.event;
 if (e.preventDefault) e.preventDefault();
 else e.returnValue = false;
 alert('Перехода по гиперссылке не будет!');
}
</script>
</head>
<body>
<form action="file.php" method="GET" onsubmit="return testEmail()">
 <div>
 E-mail:

 <input type="text" name="email" id="email">

```

```
 <input type="submit" value="Отправить">
 </div>
</form>
<p>
<a href="file.html"
 onclick="alert('Перехода по гиперссылке не будет!'); return false">
 Нажмите для перехода по гиперссылке

 Нажмите для перехода по гиперссылке
</p>
</body>
</html>
```

## 22.6. Всплывание событий

Рассмотрим пример из листинга 22.5.

### Листинг 22.5. Всплывание событий

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>Всплывание событий</title>
<script>
function showMsg(msg) {
 var div1 = document.getElementById('div1');
 div1.innerHTML += msg + '
';
}
</script>
</head>
<body onclick="showMsg('Событие onclick - Страница')">
<p onclick="showMsg('Событие onclick - Абзац')">
Щелкните мышью

здесь
</p>
<div id="div1"></div>
</body>
</html>
```

Мы написали обработчики события `onclick` для трех элементов страницы: тега `<body>`, абзаца и тега `<span>`. Попробуем щелкнуть левой кнопкой мыши на слове "здесь". В итоге вместо одного события `onclick` получим целую последовательность событий:

Событие onclick - SPAN  
 Событие onclick - Абзац  
 Событие onclick - Страница

Иными словами, событие onclick передается элементу-родителю, далее — родителю родителя и т. д. Для тега <span> элементом-родителем является абзац, а для абзаца — тег <body>. Такое прохождение событий называется *всплыванием событий*.

Управлять всплыванием события позволяют следующие свойства и методы объекта event:

- ☐ bubbles — содержит true, если текущее событие может всплывать, и false — в противном случае;
- ☐ stopPropagation() — прерывает всплывание события.

Продemonстрируем прерывание всплывания события на примере (листинг 22.6).

#### Листинг 22.6. Прерывание всплывания события

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>Прерывание всплывания события</title>
<script>
function showMsg(e, msg) {
 var div1 = document.getElementById('div1');
 div1.innerHTML += msg + '
';
 e = e || window.event;
 if (e.stopPropagation) e.stopPropagation();
 else e.cancelBubble = true;
}
</script>
</head>
<body onclick="showMsg(event, 'Событие onclick - Страница')">
<p onclick="showMsg(event, 'Событие onclick - Абзац')">
Щелкните мышью
<span style="color: red"
 onclick="showMsg(event, 'Событие onclick - SPAN')">здесь
</p>
<div id="div1"></div>
</body>
</html>
```

Попробуем теперь щелкнуть левой кнопкой мыши на слове "здесь". В итоге вместо трех событий мы получим только одно:

Событие onclick - SPAN

## 22.7. Фазы событий

Помимо фазы всплывания события существует также фаза перехвата события. Узнать текущую фазу позволяет свойство `eventPhase` объекта `event`. Оно может содержать следующие значения:

- 1 — событие было перехвачено у вложенного элемента в его родителе;
- 2 — событие в данный момент обрабатывается в том же элементе, в котором и возникло;
- 3 — событие "всплыло" из вложенного элемента.

При изучении метода `addEventListener()` мы оставили без внимания параметр `<фаза>`. Чтобы понять смысл этого параметра, рассмотрим пример (листинг 22.7).

### Листинг 22.7. Фазы событий

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>Фазы событий</title>
</head>
<body>
<div>span1
 Щелкните здесь (span2)
</div>
<script>
function handler(e) {
 console.log('Элемент ' + this.id +
 ' . Событие возникло в ' + e.target.id);
 console.log(e.eventPhase);
}
if (document.addEventListener) {
 var span1 = document.getElementById('span1');
 var span2 = document.getElementById('span2');
 span1.addEventListener('click', handler, true);
 span2.addEventListener('click', handler, false);
}
</script>
</body>
</html>
```

При щелчке на фразе "Щелкните здесь" возникнет следующая последовательность событий:

Элемент `span1`. Событие возникло в `span2`  
Элемент `span2`. Событие возникло в `span2`

Таким образом, событие, возникшее во вложенном элементе, вначале обрабатывается элементом-родителем, а затем самим элементом. Если заменить `true` на `false`, то последовательность будет другой:

Элемент `span2`. Событие возникло в `span2`

Элемент `span1`. Событие возникло в `span2`

Это нормальная последовательность всплывания событий, которую мы рассматривали в *разд. 22.6*. Именно значение `false` используется в большинстве случаев.

## 22.8. События веб-страницы

Перечислим основные события страницы:

- `onload` — возникает после полной загрузки страницы. Внутри обработчика этого события можно обращаться ко всем элементам страницы, в частности назначать обработчики возникающих в них событий. Пример:

```
window.onload = function() {
 // Здесь назначаем обработчики событий для элементов
};
```

- `onscroll` — при прокручивании содержимого элемента страницы, окна или фрейма;
- `onresize` — при изменении размеров окна;
- `onbeforeunload` — перед выгрузкой страницы. Если внутри обработчика вернуть строку, то веб-браузер выведет окно, в котором пользователю будет предложено остаться;
- `onunload` — непосредственно перед выгрузкой страницы. Генерируется после события `onbeforeunload`;
- `onbeforeprint` — перед распечаткой страницы или перед выводом ее на предварительный просмотр;
- `onafterprint` — после распечатки страницы или после вывода ее на предварительный просмотр.

Пример обработки событий страницы приведен в листинге 22.8.

### Листинг 22.8. События веб-страницы

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>События веб-страницы</title>
<script>
function showMsg(event, msg) {
 var div1 = document.getElementById('div1');
 div1.innerHTML += msg + '
';
}
```

```
 console.log(event);
}
window.onload = function() {
 showMsg(event, 'Событие onload');
};
window.onscroll = function() {
 showMsg(event, 'Событие onscroll');
};
window.onresize = function() {
 showMsg(event, 'Событие onresize');
};
window.onbeforeprint = function() {
 showMsg(event, 'Событие onbeforeprint');
};
window.onafterprint = function() {
 showMsg(event, 'Событие onafterprint');
};
window.onbeforeunload = function() {
 return 'Вы действительно хотите уйти?';
};
</script>
</head>
<body>
<div id="div1"></div>
<p>Гиперссылка</p>
<div style="height: 600px"></div>
</body>
</html>
```

Попробуйте перейти по гиперссылке или закрыть окно веб-браузера. Появится окно с запросом. Событие `onbeforeunload`, возникающее перед выгрузкой страницы, позволяет вывести стандартное диалоговое окно с двумя кнопками. Окно может содержать текст, возвращенный из обработчика, но большинство веб-браузеров выводит стандартное сообщение.

## 22.9. События мыши

Перечислим основные события мыши:

- ☐ `onclick` — возникает при щелчке мыши на элементе или на самой странице;
- ☐ `ondblclick` — при двойном щелчке мыши;
- ☐ `onmousedown` — при нажатии кнопки мыши на элементе страницы или на самой странице;
- ☐ `onmouseup` — при отпускании ранее нажатой кнопки мыши;
- ☐ `onmousemove` — при любом перемещении мыши над элементом страницы или над самой страницей;



- ❑ `onmouseover` — при наведении курсора мыши на элемент;
- ❑ `onmouseout` — при уходе курсора мыши с элемента;
- ❑ `oncontextmenu` — при нажатии правой кнопки мыши для вывода контекстного меню;
- ❑ `onwheel` — при вращении колесика мыши.

События возникают последовательно, например последовательность событий при нажатии кнопки мыши на элементе страницы будет такой:

```
onmousedown
onmouseup
onclick
```

При двойном нажатии последовательность будет такой:

```
onmousedown
onmouseup
onclick
ondblclick
```

Это значит, что событие `ondblclick` возникает после события `onclick`.

Продemonстрируем обработку событий мыши на примере (листинг 22.9).

#### Листинг 22.9. События мыши

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>События мыши</title>
<script>
function showMsg(event, msg) {
 var div1 = document.getElementById('div1');
 div1.innerHTML += msg + '
';
 console.log(event);
}
</script>
</head>
<body onmousedown="showMsg(event, 'Событие onmousedown')"
 onmouseup="showMsg(event, 'Событие onmouseup')"
 onclick="showMsg(event, 'Событие onclick')"
 ondblclick="showMsg(event, 'Событие ondblclick')"
 oncontextmenu="showMsg(event, 'Событие oncontextmenu')">
<p onmouseover="showMsg(event, 'Событие onmouseover')"
 onmouseout="showMsg(event, 'Событие onmouseout')">
Щелкните мышью в любом месте страницы
</p>
```

```
<div id="div1"></div>
<div style="height: 600px"></div>
</body>
</html>
```

Получить сведения о событии позволяют следующие свойства объекта класса Event:

- ☐ `clientX` и `clientY` — координаты курсора мыши по осям *X* и *Y* относительно левого верхнего угла внутренней области окна веб-браузера в пикселах;
- ☐ `x` и `y` — то же самое, что и `clientX` и `clientY`;
- ☐ `pageX` и `pageY` — координаты курсора мыши по осям *X* и *Y* относительно левого верхнего угла самой страницы в пикселах;
- ☐ `offsetX` и `offsetY` — координаты курсора мыши по осям *X* и *Y* относительно левого верхнего элемента страницы, в котором возникло событие, в пикселах;
- ☐ `screenX` и `screenY` — координаты курсора мыши по осям *X* и *Y* относительно левого верхнего угла экрана в пикселах;
- ☐ `button` — число, указывающее нажатую кнопку мыши. Может принимать следующие значения:
  - 0 — левая кнопка;
  - 1 — средняя кнопка или колесико;
  - 2 — правая кнопка;
  - 3 — четвертая кнопка (если она есть);
  - 4 — пятая кнопка (если она есть);
- ☐ `relatedTarget` — ссылка на элемент, с которого переместился курсор мыши (в случае события `onmouseover`) или на который переместился курсор мыши (в случае события `onmouseout`);
- ☐ `detail` — число щелчков мышью (в случае события `onclick`) или число щелчков мышью плюс 1 (в случае событий `onmousedown` и `onmouseup`);
- ☐ `deltaMode` — целочисленное обозначение единицы, в которой измеряются углы поворота колесика мыши:
  - 0 — пиксели;
  - 1 — строки;
  - 2 — страницы;
- ☐ `deltaX`, `deltaY` и `deltaZ` — величины, на которые было повернуто колесико мыши по координатным осям *X*, *Y* и *Z* соответственно.

## 22.10. События клавиатуры

Перечислим события клавиатуры:

- ☐ `onkeydown` — возникает при нажатии клавиши;
- ☐ `onkeypress` — аналогично событию `onkeydown`, но генерируется только при нажатии символьных клавиш;
- ☐ `onkeyup` — при отпуске ранее нажатой клавиши.

При нажатии клавиши последовательность событий будет такой:

```
onkeydown
onkeypress
onkeyup
```

Продemonстрируем обработку событий клавиатуры на примере (листинг 22.10).

### Листинг 22.10. События клавиатуры

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>События клавиатуры</title>
<script>
function showMsg(event, msg) {
 var div1 = document.getElementById('div1');
 div1.innerHTML += msg + '
';
 console.log(event);
}
</script>
</head>
<body onkeydown="showMsg(event, 'Событие onkeydown')"
 onkeypress="showMsg(event, 'Событие onkeypress')"
 onkeyup="showMsg(event, 'Событие onkeyup')">
<p>Нажмите клавишу на клавиатуре</p>
<div id="div1"></div>
</body>
</html>
```

Получить сведения о событии позволяют следующие свойства:

- ☐ `code` — строка с кодом нажатой клавиши (например, `'KeyD'`, `'ArrowDown'` и пр.).
- ☐ `key` — строка с символом, соответствующим клавише, или описанием клавиши (например, `'D'`, `'ArrowDown'` и пр.);

#### ПРИМЕЧАНИЕ

Список кодов клавиш можно найти на странице [https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent/code/code\\_values](https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent/code/code_values), а список описаний клавиш — на странице [https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent/key/Key\\_Values](https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent/key/Key_Values).

- `location` — целочисленное обозначение местоположения нажатой клавиши:
  - 0 — основная клавиатура;
  - 1 — левая функциональная клавиша (например, левая <Ctrl>);
  - 2 — правая функциональная клавиша (например, правая <Ctrl>);
  - 3 — дополнительная (цифровая клавиатура);
- `repeat` — `true`, если клавиша удерживается нажатой;
- `shiftKey` — `true`, если была нажата клавиша <Shift>;
- `ctrlKey` — `true`, если была нажата клавиша <Ctrl>;
- `altKey` — `true`, если была нажата клавиша <Alt>;
- `metaKey` — `true`, если была нажата клавиша <Meta> (если она присутствует на клавиатуре).

## 22.11. События аудио- и видеопроигрывателей

Аудио- и видеопроигрыватели генерируют следующие события:

- `onloadstart` — возникает, когда веб-браузер только начинает загрузку файла с роликом;
- `onloadedmetadata` — возникает после загрузки заголовка ролика, где хранятся сведения о нем;
- `ondurationchange` — возникает, когда веб-браузер получает значение продолжительности ролика или когда это значение почему-то изменяется;
- `onloadeddata` — возникает, если веб-браузер загрузил достаточно данных для вывода на экран первого кадра, но не для начала воспроизведения ролика;
- `oncanplay` — возникает, если веб-браузер получил достаточно данных, чтобы по крайней мере начать воспроизведение ролика, однако в будущем возможны приостановки воспроизведения для подгрузки данных;
- `oncanplaythrough` — возникает, если веб-браузер получил достаточно данных, чтобы начать воспроизведение ролика без приостановок для подгрузки данных;
- `onprogress` — периодически возникает в процессе загрузки мультимедийного файла;
- `onplay` — возникает после начала или возобновления воспроизведения ролика;
- `onpause` — возникает, когда воспроизведение ролика приостанавливается;
- `onended` — возникает, если воспроизведение ролика закончилось;
- `ontimeupdate` — возникает при изменении текущей позиции воспроизведения;
- `onvolumechange` — возникает при изменении значения громкости, приглушении звука или восстановлении его после приглушения;

- ❑ `onseeking` — возникает после начала изменения текущей позиции воспроизведения ролика;
- ❑ `onseeked` — возникает по завершении изменения текущей позиции воспроизведения ролика;
- ❑ `onemptied` — возникает при выгрузке файла ролика, что может произойти, например, при вызове метода `load()`;
- ❑ `onwaiting` — возникает, когда воспроизведение ролика приостанавливается для подгрузки очередной порции данных;
- ❑ `onplaying` — возникает, когда веб-браузер получает достаточно данных, чтобы возобновить воспроизведение приостановленного ролика;
- ❑ `onstalled` — возникает через три секунды после остановки подгрузки данных из мультимедийного файла. Обычно означает, что дальнейшая загрузка невозможна;
- ❑ `onerror` — возникает при прерывании загрузки файла с роликом в результате ошибки скрипта или сетевого сбоя;
- ❑ `onabort` — возникает, когда пользователь прерывает загрузку файла с роликом;
- ❑ `onratechange` — возникает при изменении значения скорости воспроизведения ролика.



## ГЛАВА 23

# Взаимодействие с элементами форм

В этой главе мы научимся с помощью JavaScript обрабатывать данные, введенные пользователем в элементы формы. Предварительная обработка введенных данных на стороне клиента позволит снизить нагрузку на веб-сервер.

### 23.1. Элементы формы

Для начала перечислим основные элементы форм:

- ❑ `<input type="text">` — текстовое поле для ввода произвольного значения;
- ❑ `<input type="password">` — поле для ввода пароля;
- ❑ `<input type="search">` — поле для ввода подстроки поиска;
- ❑ `<input type="tel">` — поле для ввода телефонного номера;
- ❑ `<input type="email">` — поле для ввода адреса электронной почты;
- ❑ `<input type="url">` — поле для ввода интернет-адреса;
- ❑ `<input type="number">` — поле для ввода целого числа с кнопками увеличения и уменьшения заносимого значения;
- ❑ `<input type="range">` — регулятор;
- ❑ `<input type="date">` — поле для ввода даты;
- ❑ `<input type="time">` — поле для ввода времени;
- ❑ `<input type="color">` — поле для выбора цвета;
- ❑ `<input type="file">` — поле для выбора файла;
- ❑ `<textarea>Текст</textarea>` — область редактирования;
- ❑ `<input type="checkbox">` — флажок;
- ❑ `<input type="radio">` — переключатель;
- ❑ `<select><option>Элемент</option></select>` — список, обычный или раскрывающийся (в зависимости от указанных параметров);
- ❑ `<input type="hidden">` — скрытое поле;

- `<input type="submit">` — кнопка отправки данных;
- `<input type="image">` — кнопка отправки данных в виде графического изображения;
- `<input type="reset">` — кнопка очистки формы;
- `<input type="button">` — обычная кнопка.

Все элементы должны быть расположены внутри парного тега `<form>`. Именно форма определяет, что делать с данными дальше. Параметр `action` тега `<form>` задает интернет-адрес серверной программы, которая будет обрабатывать введенные в форму данные, параметр `method` — метод пересылки данных серверной программе (GET или POST), а параметр `enctype` — MIME-тип передаваемых данных.

Во всех элементах формы, за исключением кнопок, с помощью параметра `name` необходимо указать название, которое пересылается на веб-сервер вместе со значением, занесенным в элемент формы. Название элемента в пределах формы должно быть уникальным, за исключением переключателей, объединенных в группу.

Название с помощью того же параметра `name` можно указать и у самой формы. Это позволит получить доступ к форме через коллекцию `forms` (будет описана далее).

Для доступа к элементам формы из скриптов в параметре `id` необходимо указать идентификатор. Обычно он совпадает с названием. Пример:

```
<input type="text" name="text1" id="text1">
```

Если данные не нужно отправлять на веб-сервер, можно вообще не использовать тег `<form>`. В этом случае вся обработка осуществляется с помощью скриптов.

## 23.2. Коллекция *forms*.

### Доступ к элементу формы из скрипта

Все формы, имеющиеся на странице, доступны через коллекцию `forms` объекта `document`. Например, чтобы получить значение текстового поля с названием `text1`, входящего в состав формы с названием `form1`, можно воспользоваться следующей строкой кода:

```
document.forms['form1'].text1.value
```

К форме также можно обратиться по индексу:

```
document.forms[0].text1.value
```

Получить доступ к элементу, вне зависимости от того, находится он внутри формы или нет, позволяет метод `getElementById()` объекта `document`. Этому методу в качестве параметра следует передать идентификатор элемента. Пример:

```
document.getElementById('text1').value
```

Все элементы формы доступны через коллекцию `elements` объекта формы:

```
document.forms['form1'].elements['text1'].value
document.forms['form1'].elements[0].value
document.forms[0].elements[0].value
```

## 23.3. Работа с формами

Объект формы поддерживает следующие свойства:

- ☐ `length` — число элементов в форме;
- ☐ `action` — интернет-адрес серверной программы, которой будут отправлены занесенные в форму данные;
- ☐ `method` — метод пересылки данных из формы серверной программе;
- ☐ `enctype` — MIME-тип передаваемых данных;
- ☐ `target` — имя фрейма, в который будет загружена страница, являющаяся результатом обработки данных формы серверной программой;
- ☐ `autocomplete` — если `'on'`, в форме будет работать автодополнение, если `'off'` — не будет работать;
- ☐ `noValidate` — если `false`, занесенные в форму данные будут проверяться на корректность перед отправкой, если `true` — не будут проверяться.

Объект формы поддерживает следующие методы:

- ☐ `submit()` — выполняет отправку данных формы серверной программе. Аналогично нажатию кнопки отправки данных;
- ☐ `reset()` — очищает форму, т. е. все элементы формы получают значения по умолчанию. Аналогично нажатию кнопки очистки;
- ☐ `checkValidity()` — возвращает `true`, если во все элементы формы занесены корректные данные, и `false` — в противном случае:

```
<form action="test.php" id="frm">
<input type="url" name="url" required>
<input type="button" value="Проверить" onclick="handler()">
</form>
<script>
function handler() {
 var frm = document.getElementById('frm');
 if (!frm.checkValidity()) {
 window.alert('Введите корректные данные!');
 }
}
</script>
```

Объект формы поддерживает следующие события:

- ☐ `onsubmit` — возникает при отправке данных формы;
- ☐ `onreset` — возникает при очистке формы.



## 23.4. Работа с элементами формы

Все элементы формы поддерживают следующие свойства:

- ❑ `required` — если `true`, то в элемент формы обязательно должно быть занесено какое-либо значение, если `false` — элемент можно оставить пустым;
- ❑ `disabled` — если `true`, элемент неактивен, если `false` — активен;
- ❑ `readOnly` — если `true`, элемент доступен только для чтения, если `false` — для чтения и ввода;
- ❑ `form` — ссылка на форму, в которой находится элемент;
- ❑ `type` — тип элемента формы;
- ❑ `autofocus` — если `true`, элемент сразу после открытия страницы получит фокус ввода, если `false` — не получит. Только один элемент на странице должен быть помечен как автоматически получающий фокус ввода;
- ❑ `validationMessage` — текст сообщения об ошибке ввода в виде строки;
- ❑ `validity` — сведения об ошибках ввода в виде объекта класса `ValidityState`, поддерживающего следующие доступные только для чтения свойства:
  - `valueMissing` — `true`, если пользователь не занес значение в элемент, помеченный как обязательный для заполнения;
  - `badInput` — `true`, если введенное посетителем значение не соответствует требуемому типу (например, в поле ввода числа занесено что-либо, отличное от целого числа);
  - `typeMismatch` — `true`, если введенное значение соответствует требуемому типу, но не укладывается в требуемый формат (например, не является адресом электронной почты);
  - `tooShort` — `true`, если введенная строка имеет длину меньшую, чем указано в параметре `minlength`;
  - `tooLong` — `true`, если введенная строка имеет длину большую, чем указано в параметре `maxlength`;
  - `patternMismatch` — `true`, если введенное значение не совпадает с заданным шаблоном (регулярным выражением, заданным в параметре `pattern`);
  - `rangeUnderflow` — `true`, если введенное число меньше указанного минимального значения;
  - `rangeOverflow` — `true`, если введенное число больше указанного максимального значения;
  - `stepMismatch` — `true`, если введенное число не укладывается в заданный интервал;
  - `customError` — `true`, если было задано иное сообщение об ошибке ввода данных (как это сделать, будет описано далее);

- `valid` — `true`, если введенное значение полностью корректно (и все перечисленные ранее свойства хранят значение `false`);
- `willValidate` — содержит `true`, если значение, присутствующее в этом элементе формы, будет проверяться на корректность. У элементов неактивных, доступных лишь для чтения, скрытых полей и кнопок содержит `false`.

Методы, поддерживаемые всеми элементами формы:

- `blur()` — убирает фокус ввода с текущего элемента формы;
- `focus()` — помещает фокус на текущий элемент формы.

События, возникающие во всех элементах формы:

- `onblur` — возникает при потере фокуса элементом формы;
- `onfocus` — возникает при получении фокуса ввода элементом формы;
- `oninvalid` — возникает, если в элемент занесены некорректные данные.

Продemonстрируем обработку некоторых событий элементов формы и самой формы на примере (листинг 23.1).

#### Листинг 23.1. События формы и ее элементов

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>События формы и ее элементов</title>
<script>
function showMsg(event, msg) {
 var div1 = document.getElementById('div1');
 div1.innerHTML += msg + '
';
 console.log(event);
}
</script>
</head>
<body>
 <form action="file.php" method="GET"
 onsubmit="showMsg(event, 'Событие onsubmit'); return false"
 onreset="showMsg(event, 'Событие onreset')">
 <div>
 Логин:

 <input type="text" name="login"
 onfocus="showMsg(event, 'Событие onfocus')"
 onblur="showMsg(event, 'Событие onblur')"
 onchange="showMsg(event, 'Событие onchange')"
 oninput="showMsg(event, 'Событие oninput')">

 E-mail:

 <input type="text" name="email" required
 oninvalid="showMsg(event, 'Событие oninvalid')">

```

```
Описание:

<textarea name="descr" rows="10" cols="15"></textarea>

<input type="reset" value="Очистить">
<input type="submit" value="Отправить">
</div>
</form>
<div id="div1"></div>
</body>
</html>
```

### 23.4.1. Текстовые поля, поля ввода пароля, подстроки поиска, адреса электронной почты, интернет-адреса и телефона

Все перечисленные в заголовке текстовые поля имеют следующие свойства:

- ☐ `value` — значение, занесенное в поле;
- ☐ `defaultValue` — изначальное значение, заданное параметром `value`;
- ☐ `minLength` — минимальное число символов, которое может быть введено в поле;
- ☐ `maxLength` — максимальное число символов, которое может быть введено в поле;
- ☐ `placeholder` — текст подсказки, выводимой прямо в поле, в виде строки;
- ☐ `pattern` — регулярное выражение, определяющее формат заносимого в поле значения и записанное в виде строки;
- ☐ `selectionStart` — номер первого символа выделенного фрагмента текста в виде числа. Если текст не выделен, хранит номер символа, на котором стоит текстовый курсор;
- ☐ `selectionEnd` — номер последнего символа выделенного фрагмента текста в виде числа. Если текст не выделен, хранит номер символа, на котором стоит текстовый курсор;
- ☐ `autocomplete` — аналогично одноименному свойству формы.

Метод `select()` выделяет весь текст в поле.

Также всеми полями поддерживаются следующие события:

- ☐ `oninput` — периодически возникает в процессе ввода данных в поле;
- ☐ `onchange` — возникает при изменении данных в поле и последующем перемещении фокуса на другой элемент формы (перед событием `onblur`);
- ☐ `onselect` — возникает при выделении фрагмента текста.

В качестве примера рассмотрим форму для занесения адреса электронной почты и пароля с проверкой правильности ввода (листинг 23.2). Если данные введены неправильно, то при отправке формы:

- ☐ поле выделяется розовым цветом;
- ☐ текст в поле выделяется;

- ☐ выводится сообщение об ошибке;
- ☐ отправка формы прерывается.

Поле **Повтор E-mail** доступно только для чтения. При вводе адреса электронной почты данные автоматически копируются из поля **E-mail** в поле **Повтор E-mail**.

**Листинг 23.2. Форма с проверкой правильности ввода**

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>Использование текстовых полей</title>
</script>
function frmSubmit() {
 var email = document.getElementById('email');
 var passwd = document.getElementById('passwd');
 email.style.backgroundColor = 'FFFFFF';
 passwd.style.backgroundColor = 'FFFFFF';
 var p = /^[a-z0-9_-]+@([a-z0-9-]+\.)+[a-z]{2,6}$/i;
 var str = email.value;
 if (!p.test(str)) {
 window.alert('Неверный адрес E-mail');
 email.style.backgroundColor = 'FFE4E1';
 email.select();
 return false;
 }
 p = /^[a-z0-9_-]{6,16}$/i;
 str = passwd.value;
 if (!p.test(str)) {
 window.alert('Неверный пароль');
 passwd.style.backgroundColor = 'FFE4E1';
 passwd.select();
 return false;
 }
 var msg = 'Вы ввели следующие данные:\n\n E-mail: ' +
 msg += email.value + '\n Пароль: ' + passwd.value;
 window.alert(msg);
 return true;
}
function frmReset() {
 document.getElementById('email').style.backgroundColor = 'FFFFFF';
 document.getElementById('passwd').style.backgroundColor = 'FFFFFF';
}
window.onload = function() {
 document.getElementById('email2').readOnly = true;
```

```
document.getElementById('email').onkeyup = function() {
 document.getElementById('email2').value =
 document.getElementById('email').value;
}
}
</script>
</head>
<body>
<form action="test.php" method="GET" name="frm" id="frm"
 onsubmit="return frmSubmit()" onreset="frmReset()">
 <div>
 E-mail:

 <input type="text" name="email" id="email"
 style="background-color: #FFFFFF">

 Повтор E-mail:

 <input type="text" name="email2" id="email2" readonly
 style="background-color: #FFFFFF">

 Пароль:

 <input type="password" name="passwd" id="passwd"
 style="background-color: #FFFFFF">

 <input type="reset" value="Очистить">
 <input type="submit" value="Отправить">
 </div>
</form>
</body>
</html>
```

## 23.4.2. Поле ввода числа и регулятор

Поле ввода числа поддерживает все свойства, методы и события, описанные в *разд. 23.4.1*. Кроме того, это поле, а также регулятор поддерживают следующие три свойства:

- `min` — минимальное значение, которое можно указать в поле или регуляторе;
- `max` — максимальное значение, которое можно указать в поле или регуляторе;
- `step` — интервал между значениями, допустимыми для занесения в поле или регулятор.

Также поддерживаются два метода:

- `stepUp([<Шаг>])` — увеличивает значение, занесенное в поле или регулятор, на заданный шаг. Если шаг не указан, берется значение, записанное в параметре `step` тега `<input>`, или, если такого параметра нет, значение 1;
- `stepDown([<Шаг>])` — уменьшает значение, занесенное в поле или регулятор, на заданный шаг. Если шаг не указан, берется значение, записанное в параметре `step` тега `<input>`, или, если такого параметра нет, значение 1.

### 23.4.3. Поля ввода даты и времени

Поля ввода даты и времени поддерживают свойства и методы, описанные в разд. 23.4.2.

Выбранная пользователем дата хранится в свойстве `value` в виде строки формата `<Год>-<Номер месяца>-<Число>`, выбранное время — в виде строки формата `<Часы>:<Минуты>`.

Значения параметров `min` и `max` тега `<input>` записываются в тех же форматах. Значение параметра `step` измеряется в днях и секундах соответственно.

Примеры:

```
<input type="date" name="enddate" min="2021-06-01" max="2021-12-31"
 step="7">
```

```
<!-- Не забываем, что значение параметра step поля ввода времени
 указывается в секундах (здесь задано значение 300 секунд == 5 минут)
-->
```

```
<input type="time" name="dinnertime" min="12:00" max="15:00" step="300">
```

### 23.4.4. Поле выбора цвета

Выбранный пользователем цвет хранится в свойстве `value` в виде строки формата `#<R><G><B>`, где `<R>`, `<G>` и `<B>` — соответственно доля красной, зеленой и синей составляющей в цвете, записанная в шестнадцатеричной системе счисления. Пример:

```
<input type="color" name="favouritecolor" value="#ff0000">
```

### 23.4.5. Поле выбора файла

Поле выбора файла поддерживает следующие свойства:

- `files` — ссылка на объект класса `FileList` — коллекцию выбранных пользователем файлов. Если в поле можно выбрать лишь один файл, коллекция будет содержать всего один элемент;
- `multiple` — если `true`, то в поле можно выбрать произвольное число файлов, если `false` — только один;
- `accept` — строка с перечнем форматов, к которым должны относиться файлы, выбираемые в поле. Отдельные форматы отделяются друг от друга запятыми. Каждый формат может быть задан в виде:
  - расширения файла — обязательно с начальной точкой;
  - MIME-типа;
  - строки `'image/*'` — любой графический файл одного из форматов, используемых в Интернете;
  - строки `'audio/*'` — любой аудиофайл одного из форматов, применяемых в Интернете;

- строки 'video/\*' — любой видеофайл одного из форматов, используемых в Интернете.

Пример:

```
<input type="file" name="docfiles" accept=".rtf,.txt,application/msword"
multiple>
```

Каждый элемент коллекции `files` представляет собой объект класса `File`, представляющий отдельный выбранный файл. Он поддерживает следующие свойства, доступные только для чтения:

- ☐ `name` — имя файла (без пути);
- ☐ `size` — размер файла в байтах;
- ☐ `type` — MIME-тип файла;
- ☐ `lastModified` — дата и время последнего изменения файла, представленные в виде числа миллисекунд, которые прошли с начала эпохи Unix.

## 23.4.6. Область редактирования

Область редактирования поддерживает те же свойства, методы и события, что и простое поле ввода (см. *разд. 23.4.1*). Кроме того, поддерживаются еще два свойства:

- ☐ `wrap` — режим переноса слов в виде одного из следующих строковых обозначений:
  - 'off' — не выполнять перенос;
  - 'soft' — слова переносятся только на экране, но не при передаче данных серверу;
  - 'hard' — слова переносятся как на экране, так и при передаче данных серверу, посредством вставки в строку символов перевода строк;
- ☐ `spellcheck` — режим проверки орфографии средствами операционной системы в виде одного из следующих строковых обозначений:
  - 'default' — выполнять проверку орфографии или нет, зависит от настроек веб-браузера;
  - 'true' — всегда проверять орфографию;
  - 'false' — никогда не проверять орфографию.

Свойство `type` в случае области редактирования хранит строку 'textarea'.

Для примера рассмотрим возможность добавления слов из текстового поля в область редактирования (листинг 23.3). Добавить слово можно с помощью кнопки **Добавить слово** или клавиши <Enter>. Так как по умолчанию нажатие клавиши <Enter> приводит к отправке данных формы, то всплывание события прерывается. При нажатии кнопки **Содержимое** выводится текущее содержимое области редактирования.

**Листинг 23.3. Добавление слов из текстового поля в область редактирования**

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>Использование области редактирования</title>
<script>
function frmSubmit() {
 var v = document.getElementById('txt1').value;
 window.alert('Текущее содержимое: \n' + v);
 return false;
}
function btnClick() {
 var txt2 = document.getElementById('txt2');
 var text = txt2.value;
 if (text != '') {
 document.getElementById('txt1').value += text + '\n';
 txt2.value = '';
 txt2.focus();
 }
 else {
 window.alert('Поле не заполнено!');
 txt2.focus();
 }
}
window.onload = function() {
 document.getElementById('txt2').onkeypress = function(e) {
 e = e || window.event;
 if (e.keyCode == 13) {
 btnClick();
 if (e.preventDefault) e.preventDefault();
 else e.returnValue = false;
 }
 }
}
</script>
</head>
<body>
 <form action="test.php" method="GET" name="frm" id="frm"
 onsubmit="return frmSubmit()">
 <div>
 Слово:

 <input type="text" name="txt2" id="txt2">

 <textarea name="txt1" id="txt1" cols="15" rows="10"></textarea>

<input type="button" value="Добавить слово"
 onclick="return btnClick()">

```



```
<input type="submit" value="Содержимое">
</div>
</form>
</body>
</html>
```

### 23.4.7. Флажок и переключатель

Флажок и переключатель имеют следующие свойства:

- `checked` — если `true`, флажок или переключатель установлен, если `false` — сброшен;
- `defaultChecked` — если `true`, флажок или переключатель изначально установлен, если `false` — изначально сброшен;
- `indeterminate` — если `true`, флажок находится в неопределенном состоянии, и `false` — в одном из определенных состояний: установленном или сброшенном.

Чтобы найти установленный переключатель, необходимо перебрать все имеющиеся в группе переключатели в цикле. Рассмотрим это на примере (листинг 23.4).

#### Листинг 23.4. Обработка флажков и переключателей

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>Использование флажков и переключателей</title>
<script>
function btnClick() {
 var msg = '';
 if (document.getElementById('check1').checked) {
 msg = 'Флажок установлен\n';
 msg += 'Значение: ' + document.getElementById('check1').value+'\n';
 }
 else {
 msg = 'Флажок снят\n';
 }
 var value1 = '';
 var count = document.frm.radiol.length;
 for (i = 0; i < count; i++) {
 if (document.frm.radiol[i].checked) {
 value1 = document.frm.radiol[i].value;
 break;
 }
 }
 if (value1 == 'male') {
 msg += 'Пол: Мужской\n';
 }
}
```

```
 else {
 msg += 'Пол: Женский\n';
 }
 window.alert(msg);
}
</script>
</head>
<body>
<form action="test.php" method="GET" name="frm" id="frm">
<div>
<input type="checkbox" name="check1" id="check1" value="yes" checked>
Текст

Укажите ваш пол:

<input type="radio" name="radio1" id="radio1" value="male"
checked>Мужской
<input type="radio" name="radio1" id="radio2" value="female">Женский

<input type="button" value="Вывести значения" onclick="btnClick()">
</div>
</form>
</body>
</html>
```

## 23.4.8. Список

Свойства списка:

- `value` — значение, записанное в первом из выбранных пунктов списка (указывается в параметре `value` тега `<select>`), или пустая строка, если ни один пункт не выбран;
- `size` — число одновременно видимых пунктов списка. Если указано значение 1, список будет раскрывающимся, если больше 1 — обычным;
- `multiple` — если `true`, в списке можно выбрать одновременно несколько пунктов, если `false` — только один пункт;
- `length` — число пунктов в списке (доступно и для записи);
- `options` — ссылка на коллекцию пунктов в списке;
- `selectedOptions` — ссылка на коллекцию выбранных пунктов списка;
- `selectedIndex` — индекс первого выбранного пункта списка или `-1`, если ни один пункт не выбран;
- `type` — тип элемента формы ('`select-multiple`', если в списке можно одновременно выбрать несколько пунктов, или '`select-one`' — если только один пункт);

Методы списка:

- `add(<Добавляемый пункт>[, <Индекс>])` — добавляет в текущий список новый пункт, указанный в первом параметре. Добавляемый пункт должен быть пред-

ставлен объектом, созданным методом `createElement()` (см. *разд. 21.9*). Вторым параметром можно указать индекс пункта, перед которым должен быть помещен добавляемый пункт. Если указан несуществующий индекс или значение `null`, новый пункт будет добавлен в конец списка;

- ❑ `remove(<Индекс>)` — удаляет из текущего списка пункт с указанным индексом. Если указан несуществующий индекс, ничего не делает.

Свойства пункта списка:

- ❑ `selected` — если `true`, пункт в данный момент выбран, если `false` — не выбран;
- ❑ `text` — текст пункта списка;
- ❑ `value` — значение пункта, заданное в параметре `value` тега `<option>`;
- ❑ `index` — индекс пункта в списке;
- ❑ `defaultSelected` — если `true`, то данный пункт выбран в списке изначально, если `false` — не выбран;
- ❑ `disabled` — если `true`, пункт неактивен, если `false` — активен.

Страница, код которой приведен в листинге 23.5, демонстрирует следующие возможности:

- ❑ добавление нового пункта в список. При заполнении первого поля и нажатии клавиши `<Enter>` фокус ввода перемещается во второе поле. При заполнении второго поля и нажатии клавиши `<Enter>` введенные значения добавляются в список. Вместо клавиши `<Enter>` можно воспользоваться кнопкой **Добавить**;
- ❑ получение значений всех выбранных в списке пунктов;
- ❑ применение взаимосвязанных списков и получение значения выбранного пункта. При выборе в первом списке какой-либо темы во втором списке будут выведены элементы, относящиеся к этой теме. При выборе во втором списке какого-либо элемента выводится сообщение со значением выбранного элемента;
- ❑ применение списков вместо гиперссылок. При выборе пункта списка загружается веб-страница, находящаяся по указанному в этом пункте интернет-адресу.

#### Листинг 23.5. Обработка списков

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>Обработка списков</title>
</head>
<body>
<form action="test.php" method="GET" name="frm" id="frm">
<!-- Добавление пункта в список -->

<script>
function addOption() {
 var txt1 = document.getElementById('txt1');
```

```
var txt2 = document.getElementById('txt2');
var select1 = document.getElementById('select1');
if (txt1.value != '' && txt2.value != '') {
 var i = select1.length++;
 select1.options[i].text = txt1.value;
 select1.options[i].value = txt2.value;
 txt1.value = '';
 txt2.value = '';
 txt1.focus();
}
else {
 window.alert('Поле не заполнено!');
 txt1.focus();
}
}

function press1(e) {
 e = e || window.event;
 if (e.keyCode == 13) {
 document.getElementById('txt2').focus();
 if (e.preventDefault) e.preventDefault();
 else e.returnValue = false;
 }
}

function press2(e) {
 e = e || window.event;
 if (e.keyCode == 13) {
 addOption();
 if (e.preventDefault) e.preventDefault();
 else e.returnValue = false;
 }
}

</script>
<div>
Добавление пункта в список:

Текст пункта:

<input type="text" name="txt1" id="txt1" onkeypress="press1(event)">

Значение пункта:

<input type="text" name="txt2" id="txt2" onkeypress="press2(event)">

<select name="select1" id="select1">
</select>

<input type="button" value="Добавить" onclick="addOption()">

<!-- Список со множественным выбором -->

<script>
function multi() {
 var msg = '';
```

```

var select2 = document.getElementById('select2');
if (select2.selectedOptions) {
 var obj = select2.selectedOptions;
 var count = obj.length;
 for (var i = 0; i < count; i++) {
 msg += obj[i].value + ' - ';
 msg += obj[i].text + '\n';
 }
}
else {
 var count = select2.length;
 for (var i = 0; i < count; i++) {
 if (select2.options[i].selected) {
 msg += select2.options[i].value + ' - ';
 msg += select2.options[i].text + '\n';
 }
 }
}
window.alert(msg);
}
</script>
Список со множественным выбором:

<select name="select2" id="select2" size="5" multiple>
<option value="1" selected>Элемент1</option>
<option value="2">Элемент2</option>
<option value="3">Элемент3</option>
<option value="4">Элемент4</option>
<option value="5">Элемент5</option>
<option value="6">Элемент6</option>
</select>

<input type="button" value="Значения списка"
 onclick="multi()">

<!-- Взаимосвязанные списки -->

<script>
var arr = [];
arr[1] = ['Тема1 Элемент1', 'Тема1 Элемент2'];
arr[2] = ['Тема2 Элемент1', 'Тема2 Элемент2', 'Тема2 Элемент3'];
var value1 = [];
value1[1] = ['1', '2'];
value1[2] = ['3', '4', '5'];
function changel() {
 var index = document.getElementById('select3').value;
 var select4 = document.getElementById('select4');
 var count = arr[index].length;
 select4.length = count;

```

```
for (i = 0; i < count; i++) {
 select4.options[i].value = value1[index][i];
 select4.options[i].text = arr[index][i];
}
}

function change2() {
 var sel = document.getElementById('select4');
 var msg = 'Значение: ' + sel.options[sel.selectedIndex].value;
 msg += '\nТекст: ' + sel.options[sel.selectedIndex].text;
 window.alert(msg);
}

</script>
Взаимосвязанные списки:

<select name="select3" id="select3" size="5" onchange="change1()">
<option value="1">Тема1</option>
<option value="2">Тема2</option>
</select>

<select name="select4" id="select4" onchange="change2()">
<option value="1" selected>Тема1 Элемент1</option>
<option value="2">Тема1 Элемент2</option>
</select>

<!-- Переход на указанный сайт -->

Переход на указанный сайт:

<select
onchange="top.location.href=this.options[this.selectedIndex].value;">
<option value="#" selected>#</option>
<option value="https://www.mail.ru/">Национальная почта Mail.ru
</option>
<option value="https://www.rambler.ru/">Рамблер</option>
</select>
</div>
</form>
</body>
</html>
```

## 23.4.9. Кнопки

У всех кнопок, за исключением графической, свойство `value` задает надпись, выводимую на кнопке.

Графическая кнопка поддерживает следующие свойства:

- ☐ `src` — интернет-адрес файла с изображением, которое будет выведено на кнопке;
- ☐ `alt` — текст замены.

В приведенном далее примере (листинг 23.6) кнопка изначально неактивна и активизируется лишь при вводе какого-либо значения в текстовое поле. При нажатии

кнопки введенное значение выводится на кнопке, после чего текстовое поле очищается и кнопка деактивируется.

#### Листинг 23.6. Обработка нажатия кнопки

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>Использование кнопок</title>
<script>
function keyUp() {
 if (document.getElementById('text1').value == '') {
 document.getElementById('button1').disabled = true;
 }
 else {
 document.getElementById('button1').disabled = false;
 }
}
function btnClick() {
 document.getElementById('button1').value =
 document.getElementById('text1').value;
 document.getElementById('text1').value = '';
 document.getElementById('button1').disabled = true;
}
</script>
</head>
<body>
 <form action="test.php" method="GET" onsubmit="return false;">
 <div>
 <input type="text" name="text1" id="text1" onkeyup="keyUp()">

 <input type="button" value="Изменить текст на кнопке"
 onclick="btnClick()" id="button1" disabled>
 </div>
 </form>
</body>
</html>
```

Кнопку также можно создать с помощью парного тега `<button>`. Содержимое этого тега будет выведено на кнопке и может быть произвольным. Тип кнопки указывается в параметре `type`: `submit` (кнопка отправки данных, тип по умолчанию, если кнопка находится в форме), `reset` (кнопка очистки формы) или `button` (обычная кнопка, тип по умолчанию, если кнопка находится вне формы).

Переделаем пример из листинга 23.6, заменив тег `<input>` тегом `<button>` (листинг 23.7).

**Листинг 23.7. Использование тега <button>**

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>Использование тега <button></title>
<script>
function keyUp() {
 if (document.getElementById('text1').value == '') {
 document.getElementById('button1').disabled = true;
 }
 else {
 document.getElementById('button1').disabled = false;
 }
}
function btnClick() {
 document.getElementById('span1').textContent =
 document.getElementById('text1').value;
 document.getElementById('text1').value = '';
 document.getElementById('button1').disabled = true;
}
</script>
</head>
<body>
<form action="test.php" method="GET" onsubmit="return false;">
<div>
<input type="text" name="text1" id="text1" onkeyup="keyUp()">

<button type="button" onclick="btnClick()" id="button1" disabled>

Tекст красного цвета
</button>
</div>
</form>
</body>
</html>
```

## 23.5. Расширенная проверка значения, занесенного в поле ввода

Элементы формы могут проверять занесенные в них значения на корректность. Если какое-либо из введенных значений некорректно, веб-браузер самостоятельно выведет возле соответствующего элемента формы всплывающее сообщение с предупреждающим текстом.

Чтобы вывести сообщение с произвольным текстом, достаточно вызвать у нужного элемента формы метод `setCustomValidity(<Текст сообщения>)`. Сразу после этого



занесенное в такой элемент значение будет помечено как некорректное, веб-браузер выведет на экран сообщение с текстом, заданным в вызове метода, а данные из формы отправлены не будут.

Чтобы пометить введенное в элемент формы значение как корректное, достаточно вызвать у элемента упомянутый ранее метод, передав ему в качестве параметра пустую строку.

Пример кода, выводящего сообщение об ошибке ввода подобным образом, приведен в листинге 23.8.

#### Листинг 23.8. Вывод произвольного сообщения об ошибке

```
<input type="text" id="txtYear">
<script>
var txtYear = document.getElementById('txtYear');
txtYear.addEventListener('input', function() {
 var n = parseInt(txtYear.value, 10);
 if (!isNaN(n)) {
 if (n < 2000) {
 txtYear.setCustomValidity(
 'Значение года не должно быть меньше 2000');
 }
 else {
 txtYear.setCustomValidity('');
 }
 }
}, false);
</script>
```



## ГЛАВА 24

# Холст: программируемая графика

Современные веб-браузеры предлагают довольно богатые возможности по программному рисованию на странице произвольной графики. Для вывода графики применяется особый элемент страницы, называемый *холстом*.

### 24.1. Тег `<canvas>`

Холст создается с помощью парного тега `<canvas>`, имеющего следующие параметры:

- ❑ `width` — ширина холста;
- ❑ `height` — высота холста.

Если параметры не указаны, создается холст с размерами 300×150 пикселей.

Пример:

```
<canvas id="cnv" width="400" height="300">
<p>Ваш веб-браузер не поддерживает элемент canvas</p>
</canvas>
```

### 24.2. Создание контекста рисования

Создав на странице сам холст, мы можем получить его *контекст рисования* — набор инструментов, используемый для рисования на холсте. Для этого предназначен метод `getContext('2d')`. Пример:

```
var oCanvas = document.getElementById('cnv');
var ctxCanvas = oCanvas.getContext('2d');
```

Контекст рисования представляется объектом класса `CanvasRenderingContext2D`. Все операции по рисованию выполняются с применением его свойств и методов, которые мы далее рассмотрим.

Получить ссылку на объект холста позволяет свойство `canvas` контекста рисования:

```
console.log(ctxCanvas.canvas.height); // 300
```

## 24.3. Заливка

Свойство `fillStyle` контекста рисования задает цвет заливки рисуемых фигур, который может быть указан в любом формате, поддерживаемом языком CSS. По умолчанию цвет заливки черный.

Для примера зададим синий цвет заливки и нарисуем прямоугольник этим цветом:

```
ctxCanvas.fillStyle = 'rgb(0, 0, 255)';
ctxCanvas.fillRect(50, 50, 200, 200);
```

А теперь укажем полупрозрачный красный цвет заливки и также нарисуем прямоугольник:

```
ctxCanvas.fillStyle = 'rgba(255, 0, 0, 0.5)';
ctxCanvas.fillRect(100, 100, 150, 150);
```

## 24.4. Контур

Цвет контура рисуемых фигур задается с помощью свойства `strokeStyle` контекста рисования. Цвет может быть указан в любом формате, поддерживаемом CSS. По умолчанию цвет контура черный.

Для примера зададим синий цвет контура и нарисуем прямоугольник этим цветом:

```
ctxCanvas.strokeStyle = 'rgb(0, 0, 255)';
ctxCanvas.strokeRect(50, 50, 200, 200);
```

А теперь укажем полупрозрачный красный цвет контура и также нарисуем прямоугольник:

```
ctxCanvas.strokeStyle = 'rgba(255, 0, 0, 0.5)';
ctxCanvas.strokeRect(100, 100, 150, 150);
```

Управлять характеристиками контура позволяют следующие свойства и методы контекста рисования:

□ `lineWidth` — толщина линий контура в пикселах (по умолчанию: 1):

```
ctxCanvas.lineWidth = 20;
ctxCanvas.strokeRect(50, 50, 200, 200);
```

□ `lineCap` — форма концов у линии контура в виде одного из следующих строковых обозначений:

- `'square'` — квадратные концы (прибавляются к длине линии);
- `'butt'` — концы никак не оформляются (значение по умолчанию);
- `'round'` — закругленные концы (прибавляются к длине линии).

Пример рисования прямой линии с закругленными концами:

```
ctxCanvas.beginPath();
ctxCanvas.lineWidth = 10;
```

```
ctxCanvas.lineCap = 'round';
ctxCanvas.moveTo(20, 20);
ctxCanvas.lineTo(180, 20);
ctxCanvas.stroke();
```

□ **lineJoin** — форма соединений линий контура в виде одного из следующих строковых обозначений:

- 'miter' — обычные углы; значение по умолчанию;
- 'bevel' — скошенные углы;
- 'round' — закругленные углы.

Пример рисования треугольника со скругленными углами:

```
ctxCanvas.beginPath();
ctxCanvas.lineWidth = 15;
ctxCanvas.lineJoin = 'round';
ctxCanvas.moveTo(200, 20);
ctxCanvas.lineTo(20, 100);
ctxCanvas.lineTo(200, 100);
ctxCanvas.closePath();
ctxCanvas.stroke();
```

□ **miterLimit** — задает ограничение длины угла, когда свойству **lineJoin** дано значение 'miter', в пикселах. Если это ограничение превышено, угол будет скошенным. Значение по умолчанию: 10;

□ **setLineDash(<Значения>)** — задает значения для рисования пунктирного контура. Значения указываются в виде массива: элементы с четными индексами задают длину штриха, а элементы с нечетными индексами — длину пропуска между штрихами. Пример рисования пунктирной линии:

```
ctxCanvas.beginPath();
ctxCanvas.lineWidth = 5;
ctxCanvas.lineDashOffset = 10;
ctxCanvas.setLineDash([15.0, 10.0]);
ctxCanvas.miterLimit = 20;
ctxCanvas.moveTo(200, 20);
ctxCanvas.lineTo(50, 200);
ctxCanvas.stroke();
```

Чтобы сделать следующую линию сплошной, достаточно передать методу **setLineDash()** пустой массив;

- **lineDashOffset** — смещение начала пунктирного контура (значение по умолчанию: 0);
- **getLineDash()** — возвращает массив со значениями для рисования пунктирного контура.

## 24.5. Рисование прямоугольников

Для вывода прямоугольников предназначены следующие методы контекста рисования:

- `fillRect()` — рисует сплошной прямоугольник, используя характеристики заливки. Формат метода:

```
fillRect(<X>, <Y>, <Ширина>, <Высота>)
```

Параметры `<X>` и `<Y>` задают координаты левого верхнего угла прямоугольника. Положительная ось `x` направлена вправо, а положительная ось `y` — вниз. Пример:

```
ctxCanvas.fillStyle = 'rgb(0, 0, 255)';
ctxCanvas.fillRect(50, 50, 200, 200);
```

- `strokeRect()` — рисует контурный прямоугольник, используя характеристики контура. Формат метода:

```
strokeRect(<X>, <Y>, <Ширина>, <Высота>)
```

Параметры `<X>` и `<Y>` задают координаты левого верхнего угла прямоугольника. Положительная ось `x` направлена вправо, а положительная ось `y` — вниз. Пример:

```
ctxCanvas.strokeStyle = 'rgb(0, 0, 255)';
ctxCanvas.strokeRect(50, 50, 200, 200);
```

## 24.6. Очистка области холста

Метод `clearRect()` контекста рисования очищает заданную прямоугольную область холста от любой присутствовавшей там графики. Формат метода:

```
clearRect(<X>, <Y>, <Ширина>, <Высота>)
```

Параметры аналогичны таковым у метода `fillRect()`. Пример рисования сплошного прямоугольника и очистки прямоугольной области внутри него:

```
ctxCanvas.fillRect(0, 0, 400, 300);
ctxCanvas.clearRect(100, 100, 200, 100);
```

Пример рисования сплошного прямоугольника, очистки всего холста и рисования другого сплошного прямоугольника:

```
ctxCanvas.fillRect(0, 0, 400, 300);
ctxCanvas.clearRect(0, 0, ctxCanvas.canvas.width,
 ctxCanvas.canvas.height);
ctxCanvas.fillRect(50, 50, 100, 100);
```

## 24.7. Вывод текста

Свойство `font` контекста рисования задает параметры шрифта, которым будет выводиться текст. Эти параметры указывают в том же формате, что и у значения атрибута CSS `font`, в виде строки. Пример:

```
ctxCanvas.font = 'italic 16pt Verdana';
ctxCanvas.strokeText('JavaScript', 200, 50, 100);
```

Для вывода текста предназначены следующие методы контекста рисования:

- `strokeText()` — выводит контурный текст, используя характеристики контура. Формат метода:

```
strokeText(<Текст>, <X>, <Y>[, <Максимальная ширина>])
```

Второй и третий параметры указывают координаты точки, в которой будет выведен текст. Если указана максимальная ширина и выводимый текст получается шире, то веб-браузер выведет его либо шрифтом с уменьшенной шириной символов (если данный шрифт поддерживает такое начертание), либо шрифтом меньшего размера. Пример:

```
ctxCanvas.strokeStyle = 'blue';
ctxCanvas.font = 'italic 16pt Verdana';
ctxCanvas.strokeText('JavaScript', 200, 50, 50);
```

- `fillText()` — выводит сплошной текст, используя характеристики заливки. Формат метода:

```
fillText(<Текст>, <X>, <Y>[, <Максимальная ширина>])
```

Параметры аналогичны таковым у метода `strokeText()`. Пример:

```
ctxCanvas.fillStyle = 'blue';
ctxCanvas.font = 'italic 16pt Verdana';
ctxCanvas.fillText('JavaScript', 200, 50);
```

Помимо того, контекст рисования поддерживает еще два свойства:

- `textAlign` — горизонтальное выравнивание текста относительно точки, в которой он будет выведен, в виде одного из следующих строковых обозначений:
  - `'left'` — выравнивание по левому краю;
  - `'right'` — по правому краю;
  - `'start'` — по левому краю, если текст выводится по направлению слева направо, и по правому краю — в противном случае (значение по умолчанию);
  - `'end'` — по правому краю, если текст выводится по направлению слева направо, и по левому краю — в противном случае;
  - `'center'` — по центру.

Пример:

```
ctxCanvas.textAlign = 'center';
ctxCanvas.fillText('HTML, CSS', 100, 100);
```

□ `textBaseline` — вертикальное выравнивание выводимого текста относительно точки, в которой он будет выведен, в виде одного из следующих строковых обозначений:

- `'top'` — выравнивание по верху прописных букв;
- `'hanging'` — по верху строчных букв;
- `'middle'` — по средней линии строчных букв;
- `'alphabetic'` — по базовой линии букв европейских алфавитов (значение по умолчанию);
- `'ideographic'` — по базовой линии иероглифических символов (она находится чуть ниже базовой линии букв европейских алфавитов);
- `'bottom'` — по низу букв.

Пример:

```
ctxCanvas.textBaseline = 'bottom';
ctxCanvas.fillText('HTML, CSS', 100, 100);
```

Еще может пригодиться метод `measureText(<Текст>)`, позволяющий узнать ширину текста. Метод возвращает объект с единственным свойством `width`, которое и хранит ширину текста в пикселах. Пример:

```
var s = 'HTML, CSS, JavaScript, PHP, MySQL';
ctxCanvas.font = 'bold 24pt Tahoma';
console.log(ctxCanvas.measureText(s).width);
```

## 24.8. Вывод графических изображений

Вывести изображение на холст позволяет метод `drawImage()`. Форматы метода:

```
drawImage(<Изображение>, <x>, <y>)
drawImage(<Изображение>, <x>, <y>, <w>, <h>)
drawImage(<Изображение>, <sx>, <sy>, <sw>, <sh>,
 <dx>, <dy>, <dw>, <dh>)
```

В первом параметре можно указать собственно изображение (тег `<img>`), видеопроигрыватель или другой холст.

Первый формат выводит изображение целиком в позицию с координатами `<x>` и `<y>`:

```
<canvas id="cnv" width="400" height="300"></canvas>
<canvas id="cnv2" width="400" height="300"></canvas>

<script>
var oCanvas = document.getElementById('cnv');
var ctxCanvas = oCanvas.getContext('2d');
```

```
var imgSample = new Image();
imgSample.src = 'photo.jpg';
ctxCanvas.drawImage(imgSample, 0, 0);

var oCanvas2 = document.getElementById('cnv2');
var ctxCanvas2 = oCanvas2.getContext('2d');
ctxCanvas2.fillStyle = 'rgb(0, 0, 255)';
ctxCanvas2.fillRect(50, 50, 200, 200);
ctxCanvas.drawImage(oCanvas2, 0, 0);

var oImage = document.getElementById('img');
ctxCanvas2.drawImage(oImage, 0, 0);
</script>
```

Второй формат ограничивает область вывода шириной `<w>` и высотой `<h>`. Изображение может быть уменьшено или увеличено таким образом, чтобы вписаться в указанные размеры. Если пропорции области и изображения не совпадают, то изображение будет соответственно растянуто или сжато. Пример:

```
var img = new Image();
img.src = 'photo.jpg';
ctxCanvas.drawImage(img, 0, 0, 400, 100);
```

Третий формат берет из изображения прямоугольную область с координатами левого верхнего угла `<sx>` и `<sy>`, шириной `<sw>`, высотой `<sh>` и вписывает ее в прямоугольную область холста с координатами `<dx>` и `<dy>`, шириной `<dw>` и высотой `<dh>`:

```
var img = new Image();
img.src = 'photo.jpg';
ctxCanvas.drawImage(img, 0.0, 0.0, 50.0, 50.0,
 200.0, 20.0, 50.0, 50.0);
ctxCanvas.drawImage(img, 0.0, 0.0, 50.0, 50.0,
 200.0, 100.0, 100.0, 100.0);
```

## 24.9. Рисование сложных фигур

Для рисования сложных фигур применяются следующие методы:

- `beginPath()` — начинает рисование новой фигуры;
- `moveTo(<x>, <y>)` — перемещает текущую позицию рисования в точку с координатами `<x>` и `<y>`;
- `closePath()` — замыкает контур рисуемой фигуры;
- `stroke()` — рисует контурную фигуру, используя характеристики контура;
- `fill()` — рисует сплошную фигуру, используя характеристики заливки.



Пример рисования замкнутого треугольника с контуром и заливкой:

```
ctxCanvas.fillStyle = 'rgb(0, 0, 255)';
ctxCanvas.strokeStyle = 'rgb(255, 0, 0)';
ctxCanvas.lineWidth = 3;
ctxCanvas.beginPath();
ctxCanvas.moveTo(20, 20);
ctxCanvas.lineTo(100, 20);
ctxCanvas.lineTo(100, 80);
ctxCanvas.closePath();
ctxCanvas.stroke();
ctxCanvas.fill();
```

- `lineTo(<x>, <y>)` — добавляет прямую линию, проведенную от текущей позиции рисования до точки с указанными координатами, к текущей фигуре:

```
ctxCanvas.lineTo(100, 20);
```

- `arcTo()` — добавляет дугу к текущей фигуре. Формат метода:

```
arcTo(<x1>, <y1>, <x2>, <y2>, <radius>)
```

Первые два параметра указывают координаты первой опорной точки дуги, третий и четвертый — координаты второй опорной точки, пятый — радиус дуги в пикселах. Пример рисования дуги:

```
ctxCanvas.strokeStyle = 'rgb(0, 0, 0)';
ctxCanvas.lineWidth = 5;
ctxCanvas.beginPath();
ctxCanvas.moveTo(100, 100);
ctxCanvas.arcTo(200, 0, 300, 100, 100);
ctxCanvas.stroke();
```

- `arc()` — добавляет дугу к текущей фигуре. Формат метода:

```
arc(<centerX>, <centerY>, <radius>, <startAngle>, <endAngle>,
 <true | false>)
```

Параметры `<centerX>` и `<centerY>` задают координаты центра круга, параметр `<radius>` — радиус в пикселах, параметр `<startAngle>` — начальный угол в радианах, а параметр `<endAngle>` — конечный угол в радианах. Если шестой параметр имеет значение `true`, дуга рисуется против часовой стрелки, а если `false` — по часовой стрелке.

Пересчитать величины углов из градусов в радианы можно с помощью следующего выражения:

```
radians = (Math.PI / 180) * degrees;
```

Здесь переменная `degrees` хранит значение угла в градусах, а переменная `radians` будет хранить то же значение, но в радианах.

Пример рисования круга:

```
ctxCanvas.strokeStyle = 'rgb(0, 0, 0)';
ctxCanvas.lineWidth = 5;
```

```
ctxCanvas.beginPath();
ctxCanvas.arc(200, 150, 100, 0, Math.PI * 2, false);
ctxCanvas.stroke();
```

- ❑ `bezierCurveTo()` — добавляет кубическую кривую Безье к текущей фигуре. **Формат метода:**

```
bezierCurveTo(<xc1>, <yc1>, <xc2>, <yc2>, <x1>, <y1>)
```

Параметры `<xc1>` и `<yc1>` задают координаты первой опорной точки, параметры `<xc2>` и `<yc2>` — координаты второй опорной точки, а параметры `<x1>` и `<y1>` — координаты конечной точки. Пример:

```
ctxCanvas.beginPath();
ctxCanvas.moveTo(100, 100);
ctxCanvas.bezierCurveTo(120, 80, 160, 20, 100, 200);
ctxCanvas.stroke();
```

- ❑ `quadraticCurveTo()` — добавляет квадратичную кривую Безье к текущей фигуре. **Формат метода:**

```
quadraticCurveTo(<xc>, <yc>, <x1>, <y1>)
```

Параметры `<xc>` и `<yc>` задают координаты опорной точки, а параметры `<x1>` и `<y1>` — координаты конечной точки. Пример:

```
ctxCanvas.beginPath();
ctxCanvas.moveTo(100, 100);
ctxCanvas.quadraticCurveTo(200, 100, 200, 200);
ctxCanvas.stroke();
```

- ❑ `rect()` — добавляет к фигуре прямоугольник. **Формат метода:**

```
rect(<x>, <y>, <w>, <h>)
```

Первые два параметра задают координаты левого верхнего угла прямоугольника, третий и четвертый — его ширину и высоту соответственно. Нарисуем фигуру, состоящую из трех накладывающихся друг на друга разноцветных квадратов:

```
ctxCanvas.beginPath();
ctxCanvas.fillStyle = 'red';
ctxCanvas.rect(50, 50, 50, 50);
ctxCanvas.fill();
ctxCanvas.beginPath();
ctxCanvas.fillStyle = 'green';
ctxCanvas.rect(75, 75, 50, 50);
ctxCanvas.fill();
ctxCanvas.beginPath();
ctxCanvas.fillStyle = 'blue';
ctxCanvas.rect(100, 100, 50, 50);
ctxCanvas.fill();
```

- ❑ `clip()` — создает маску на основе контура рисуемой фигуры (контур должен быть закрытым). Область внутри контура будет видна, а вне — скрыта. Пример:

```
ctxCanvas.beginPath();
ctxCanvas.moveTo(100, 150);
ctxCanvas.lineTo(200, 0);
ctxCanvas.lineTo(200, 300);
ctxCanvas.closePath();
ctxCanvas.clip();
ctxCanvas.fillRect(0, 100, 400, 100);
```

## 24.10. Определение вхождения точки в состав контура

Выяснить, входит ли точка с заданными координатами в состав контура рисуемой фигуры, можно с помощью метода `isPointInPath(<x>, <y>)`. Метод возвращает `true`, если точка входит в состав контура, и `false` — в противном случае. Пример:

```
ctxCanvas.beginPath();
ctxCanvas.rect(50, 50, 50, 50);
ctxCanvas.stroke();
if (ctxCanvas.isPointInPath(60, 50)) {
 window.alert('Точка входит в состав контура');
}
```

## 24.11. Задание сложных цветов

Помимо однотонных цветов холст позволяет закрашивать линии и заливки градиентами и текстурами, созданными на основе заданных графических изображений.

### 24.11.1. Линейный градиент

Линейный градиент создают в три этапа.

Первый этап: собственно создание линейного градиента вызовом метода `createLinearGradient()`. Формат метода:

```
createLinearGradient(<x1>, <y1>, <x2>, <y2>)
```

В первых двух параметрах указывают горизонтальную и вертикальную координаты начальной точки градиента, в третьем и четвертом параметрах — координаты его конечной точки. Метод возвращает созданный линейный градиент, представленный объектом класса `CanvasGradient`.

Второй этап: расстановка ключевых точек градиента. Здесь нам понадобится метод `addColorStop()` класса `CanvasGradient`:

```
<Градиент>.addColorStop(<Положение ключевой точки>, <Цвет>)
```

В первом параметре указывают местоположение на градиенте ключевой точки — точки, которая должна быть закрашена цветом, указанным во втором параметре.

Положение ключевой точки задается в виде вещественного числа от 0.0 (начало градиента) до 1.0 (конец градиента), цвет — в любом формате, поддерживаемом CSS, в виде строки. Результат метод не возвращает.

Третий этап: использование готового линейного градиента. Для этого представляющий его объект класса `CanvasGradient` следует присвоить свойству `strokeStyle` или `fillStyle`.

Пример:

```
var lgSample = ctxCanvas.createLinearGradient(0, 0, 100, 50);
lgSample.addColorStop(0, 'black');
lgSample.addColorStop(0.4, 'rgba(0, 0, 255, 0.5)');
lgSample.addColorStop(1, '#ff0000');
ctxCanvas.fillStyle = lgSample;
ctxCanvas.fillRect(0, 0, 200, 100);
```

## 24.11.2. Радиальный градиент

Радиальный градиент также создают в три этапа.

Первый этап: создание радиального градиента вызовом метода `createRadialGradient()`:

```
createRadialGradient(
 <Горизонтальная координата центра внутренней окружности>,
 <Вертикальная координата центра внутренней окружности>,
 <Радиус внутренней окружности>,
 <Горизонтальная координата центра внешней окружности>,
 <Вертикальная координата центра внешней окружности>,
 <Радиус внешней окружности>)
```

Параметры этого метода определяют координаты центров и радиусы обеих окружностей, между которыми будет располагаться создаваемый радиальный градиент. Они задаются в виде чисел в пикселах. Метод возвращает объект класса `CanvasGradient`, представляющий созданный градиент.

Второй этап: расстановка ключевых точек. Он также выполняется с помощью метода `addColorStop()` класса `CanvasGradient` (см. *разд. 24.11.1*). Только в данном случае первый параметр определит относительное положение создаваемой ключевой точки на промежутке между внутренней и внешней окружностями.

Третий этап: использование созданного градиента. Он выполняется способом, описанным в *разд. 24.11.1*.

Пример:

```
var rgSample = ctxCanvas.createRadialGradient(100, 100, 10, 150, 100, 120);
rgSample.addColorStop(0, '#cccccc');
rgSample.addColorStop(0.8, 'black');
rgSample.addColorStop(1, '#00ff00');
ctxCanvas.fillStyle = rgSample;
ctxCanvas.fillRect(0, 0, 200, 100);
```

### 24.11.3. Заливка текстурой

Заливка текстурой выполняется с помощью метода `createPattern()`:

```
createPattern(<Источник>, <Режим повторения>)
```

Первый параметр задает источник текстуры в виде объекта класса `Image`, тега `<img>`, `<video>` или другого холста.

Часто бывает так, что размеры текстуры меньше, чем у закрашиваемой ею фигуры. В этом случае текстура повторяется столько раз, чтобы полностью "вымостить" контур или заливку. Режим такого повторения задает второй параметр метода `createPattern()`. Его значение должно быть одной из следующих строк:

- ☐ `'repeat'` или `''` (пустая строка, также можно указать `null`) — повтор по горизонтали и вертикали;
- ☐ `'repeat-x'` — повтор только по горизонтали;
- ☐ `'repeat-y'` — повтор только по вертикали;
- ☐ `'no-repeat'` — без повтора.

Метод `createPattern()` возвращает объект класса `CanvasPattern`.

Пример:

```
var imgSample = new Image();
imgSample.src = 'graphic_color.jpg';
var cpSample = ctxCanvas.createPattern(imgSample, 'repeat');
ctxCanvas.fillStyle = cpSample;
ctxCanvas.fillRect(0, 0, 200, 100);
```

Текстура не фиксируется на холсте, а полностью применяется к рисуемой фигуре. В этом принципиальное отличие от градиентов.

## 24.12. Сохранение и восстановление состояния

Значения основных характеристик можно сохранить в стеке, выполнить какую-либо операцию рисования, а затем восстановить эти значения из стека. Для этого предназначены следующие методы:

- ☐ `save()` — сохраняет значения основных характеристик, а именно:
  - значения свойств `globalAlpha`, `globalCompositeOperation`, `fillStyle`, `lineCap`, `lineJoin`, `lineWidth`, `miterLimit` и `strokeStyle`;
  - все заданные трансформации;
  - все заданные маски;
- ☐ `restore()` — восстанавливает значения основных характеристик из стека.

## 24.13. Преобразования

Выполнить различные преобразования системы координат холста позволяют следующие методы:

- `scale(<dx>, <dy>)` — изменяет масштаб координат холста. Параметры этого метода задают масштаб для горизонтальной и вертикальной осей системы координат в виде вещественных чисел. Числа меньше 1.0 задают уменьшение масштаба, а числа больше 1.0 — увеличение; если нужно оставить масштаб по какой-то из осей неизменным, достаточно указать значение 1.0 соответствующего параметра. Пример увеличения масштаба в два раза:

```
ctxCanvas.save();
ctxCanvas.scale(2.0, 2.0);
ctxCanvas.fillRect(10, 40, 40, 40);
ctxCanvas.restore();
```

- `rotate(<Угол в радианах>)` — поворачивает систему координат на заданный угол. При указании положительного значения угла выполняется поворот по часовой стрелке, при указании отрицательного значения — против часовой стрелки. Поворот всегда выполняется относительно начала системы координат. Пример:

```
ctxCanvas.save();
ctxCanvas.translate(200, 150);
for (var i = 0; i < 3; i++) {
 ctxCanvas.rotate(Math.PI / 6);
 ctxCanvas.strokeRect(-50, -50, 100, 100);
}
ctxCanvas.restore();
```

- `translate(<tx>, <ty>)` — сдвигает начало системы координат с точку с заданными координатами.

## 24.14. Управление наложением графики

Если нарисовать какую-либо фигуру поверх уже существующей, то новая фигура будет наложена на старую, перекрыв ее. Это поведение по умолчанию, которое можно изменить, указав другое значение для свойства `globalCompositeOperation`. Вот перечень поддерживаемых значений:

- `'source-over'` — новая фигура накладывается на старую, перекрывая ее (значение по умолчанию);
- `'source-atop'` — отображается только та часть новой фигуры, которая накладывается на старую; остальная часть новой фигуры не выводится. Старая фигура выводится целиком и находится ниже новой;
- `'source-in'` — отображается только та часть новой фигуры, которая накладывается на старую. Остальные части новой и старой фигур не выводятся;

- ❑ 'source-out' — отображается только та часть новой фигуры, которая не накладывается на старую. Остальные части новой фигуры и вся старая фигура не выводятся;
- ❑ 'destination-over' — новая фигура перекрывается старой;
- ❑ 'destination-atop' — отображается только та часть старой фигуры, которая накладывается на новую; остальная часть старой фигуры не выводится. Новая фигура выводится целиком и находится ниже старой;
- ❑ 'destination-in' — отображается только та часть старой фигуры, на которую накладывается новая. Остальные части новой и старой фигур не выводятся;
- ❑ 'destination-out' — отображается только та часть старой фигуры, на которую не накладывается новая. Остальные части новой фигуры и вся старая фигура не выводятся;
- ❑ 'xor' — отображаются только те части старой и новой фигур, которые не накладываются друг на друга;
- ❑ 'lighter' — цвета накладываемых частей старой и новой фигур складываются, результирующий цвет получается более светлым, окрашиваются накладываемые части фигур;
- ❑ 'copy' — выводится только новая фигура; все старые фигуры удаляются с холста.

Заданный нами способ наложения действует только для графики, которую мы нарисуем после этого. На уже нарисованную графику он не влияет.

Пример:

```
ctxCanvas.fillStyle = 'blue';
ctxCanvas.fillRect(0, 50, 400, 200);
ctxCanvas.fillStyle = 'red';
ctxCanvas.globalCompositeOperation = 'source-over';
ctxCanvas.fillRect(100, 0, 200, 300);
```

Этот код рисует два накладываемых прямоугольника разных цветов и позволяет изучить эффект при разных значениях свойства `globalCompositeOperation`. Изменяем значение этого свойства, перезагружаем страницу и смотрим, что получится.

## 24.15. Задание уровня прозрачности

Свойство `globalAlpha` указывает уровень прозрачности для любой графики, которую мы впоследствии нарисуем. Он задается в виде вещественного числа от 0.0 (полностью прозрачный) до 1.0 (полностью непрозрачный; значение по умолчанию). Пример:

```
ctxCanvas.globalAlpha = 0.1;
```

## 24.16. Создание тени

Для создания теней у всех рисуемых фигур следует указать параметры тени, воспользовавшись следующими четырьмя свойствами:

- `shadowOffsetX` — смещение тени по горизонтали относительно фигуры (в виде числа в пикселах; значение по умолчанию — 0);
- `shadowOffsetY` — смещение тени по вертикали относительно фигуры (в виде числа в пикселах; значение по умолчанию — 0);
- `shadowBlur` — степень размытия тени в виде вещественного числа; чем больше это число, тем сильнее размыта тень (значение по умолчанию — 0.0, т. е. отсутствие размытия);
- `shadowColor` — цвет тени в любом формате, поддерживаемом CSS, в виде строки (по умолчанию — черный непрозрачный).

Пример:

```
ctxCanvas.shadowOffsetX = 2;
ctxCanvas.shadowOffsetY = -2;
ctxCanvas.shadowBlur = 4;
ctxCanvas.shadowColor = 'rgba(128, 128, 128, 0.5)';
ctxCanvas.fillText('Двое: я и моя тень.', 150, 50);
```

## 24.17. Работа с отдельными пикселями

И наконец, мы можем работать с отдельными пикселями графики, нарисованной на холсте. Это может пригодиться при создании очень сложного изображения или наложении на графику специальных эффектов наподобие размытия.

### 24.17.1. Получение массива пикселей

Получить массив пикселей, представляющий нарисованную на холсте графику или ее фрагмент, позволяет метод `getImageData()`. Формат метода:

```
getImageData(<x>, <y>, <Ширина>, <Высота>)
```

Первые два параметра указывают координату левого верхнего угла фрагмента нарисованной графики, который будет представлять получаемый массив пикселей, два последних — ширину и высоту этого фрагмента.

Метод `getImageData()` возвращает объект класса `ImageData`, представляющий массив пикселей.

Пример:

```
var idSample = ctxCanvas.getImageData(200, 150, 100, 100);
```



## 24.17.2. Создание пустого массива пикселей

Чтобы самостоятельно нарисовать какое-либо изображение, следует сначала создать пустой массив пикселей, вызвав метод `createImageData()`:

```
createImageData(<Ширина>, <Высота>)
```

Параметры указывают ширину и высоту рисуемого изображения. Метод возвращает объект класса `ImageData`.

Пример создания пустого массива пикселей тех же размеров, что и сам холст:

```
var idEmpty = ctxCanvas.createImageData(400, 300);
```

## 24.17.3. Манипуляция пикселями

Получив массив пикселей, мы можем начать работать с отдельными пикселями этого массива, задавая для них цвет, значения полупрозрачности и тем самым формируя какое-либо изображение или изменяя уже существующее.

Класс `ImageData` поддерживает свойство `data`. Его значением является коллекция, хранящая набор чисел:

- ☐ первое число представляет собой долю красного цвета в цвете первого пикселя массива;
- ☐ второе число — долю зеленого цвета в цвете первого пикселя;
- ☐ третье число — долю синего цвета в цвете первого пикселя;
- ☐ четвертое число — степень полупрозрачности цвета первого пикселя;
- ☐ пятое число — долю красного цвета в цвете второго пикселя;
- ☐ шестое число — долю зеленого цвета в цвете второго пикселя;
- ☐ седьмое число — долю синего цвета в цвете второго пикселя;
- ☐ восьмое число — степень полупрозрачности цвета второго пикселя;
- ☐ ...
- ☐  $(n-3)$ -е число — долю красного цвета в цвете последнего пикселя;
- ☐  $(n-2)$ -е число — долю зеленого цвета в цвете последнего пикселя;
- ☐  $(n-1)$ -е число — долю синего цвета в цвете последнего пикселя;
- ☐  $n$ -е число — степень полупрозрачности цвета последнего пикселя.

Все значения, включая степень полупрозрачности, должны укладываться в диапазон от 0 до 255. Для степени полупрозрачности значение 0 задает полную прозрачность, а 255 — полную непрозрачность.

Нумерация пикселей в массиве идет слева направо и сверху вниз, т. е. по строкам.

Для доступа к отдельным значениям полученной коллекции можно использовать тот же синтаксис, что и в случае обычных массивов. Свойство `length` коллекции хранит ее размер.

В пустой коллекции все пиксели будут хранить значения 0, т. е. иметь прозрачный черный цвет.

Для примера зададим у первого (левого верхнего) пиксела полностью непрозрачный красный цвет:

```
var idEmpty = ctxCanvas.createImageData(400, 300);
idEmpty.data[0] = 255;
idEmpty.data[1] = 0;
idEmpty.data[2] = 0;
idEmpty.data[3] = 255;
ctxCanvas.putImageData(idEmpty, 0, 0);
```

## 24.17.4. Вывод массива пикселей

Завершив формирование нового изображения в массиве пикселей, мы можем вывести его на холст, вызвав метод `putImageData()`:

```
putImageData(<Массив пикселей>, <x1>, <y1>[, <x2>, <y2>, <Ширина>, <Высота>])
```

Второй и третий параметры задают координату точки, где будет находиться левый верхний угол выводимого массива пикселей.

Четвертый и пятый параметры задают координаты точки, где находится левый верхний угол фрагмента массива пикселей, который должен быть выведен на холсте, а шестой и седьмой — ширину и высоту выводимого фрагмента. Если эти параметры не указаны, будет выведено все содержимое массива пикселей.

Пример:

```
ctxCanvas.putImageData(idEmpty, 0, 0);
```

В листинге 24.1 приведен код, рисующий на странице прямоугольник с градиентной заливкой. В заливке зеленый цвет плавно сменяется синим, а потом — красным; при этом цвета становятся все более и более прозрачными.

### Листинг 24.1. Вывод массива пикселей

```
<!doctype html>
<html lang="ru">
 <head>
 <meta charset="utf-8">
 <title>Манипуляция пикселями</title>
 <script>
 window.addEventListener('load', function() {
 var oCanvas = document.getElementById('cnv');
 var ctxCanvas = oCanvas.getContext('2d');
 var idEmpty = ctxCanvas.createImageData(255, 255);
 for (var i = 0; i < idEmpty.data.length; i += 4) {
 idEmpty.data[i + 0] = i / 4 / 255;
 idEmpty.data[i + 1] = 255 - i / 4 / 255;
```

```
 idEmpty.data[i + 2] = i / 4 / 255;
 idEmpty.data[i + 3] = 255 - i / 4 / 255;
 }
 ctxCanvas.putImageData(idEmpty, 0, 0);
}, false);
</script>
</head>
<body>
 <canvas id="cnv" width="400" height="300"></canvas>
</body>
</html>
```



## ГЛАВА 25

# АJAX: обмен данными без перезагрузки веб-страницы

*AJAX* (Asynchronous JavaScript And XML, асинхронный JavaScript и XML) — это загрузка данных веб-браузером с сервера без перезагрузки страницы. Веб-браузер в фоновом режиме отправляет клиентский запрос серверной программе. Последняя обрабатывает полученный запрос и отправляет серверный ответ в виде XML-документа, фрагмента HTML-кода, скрипта JavaScript, объекта JSON или в виде простого текста. Полученный веб-браузером ответ обрабатывается специальным скриптом, который на основе этого ответа обновляет содержимое страницы. Сама страница при этом не перезагружается.

## 25.1. Основы технологии AJAX

Классический пример технологии AJAX демонстрирует поисковая система Google, позволяющая при наборе поисковой фразы в строке запроса предлагать различные варианты, начинающиеся с набранных букв (рис. 25.1).

Прежде чем изучать основы этой замечательной технологии, рассмотрим альтернативный способ "общения" веб-браузера с сервером без перезагрузки веб-страницы.

### **ВНИМАНИЕ!**

Технология AJAX позволяет загружать данные исключительно с веб-сервера. Для тестирования примеров из книги нужно установить на компьютер одну из программ: ХАМРР, Денвер или Open Server. Для запуска серверных программ можно также воспользоваться виртуальным хостингом с поддержкой PHP.

### 25.1.1. Обмен данными с помощью тега `<iframe>`

После заполнения формы и щелчка на кнопке отправки данные будут отправлены серверной программе и полученный от нее ответ полностью заменит текущую веб-страницу. Однако если в параметре `target` тега `<form>` указать название фрейма, то полученный ответ будет выведен в этом фрейме, а сама страница не перезагрузится. Так что фрейм можно рассматривать как альтернативу AJAX.

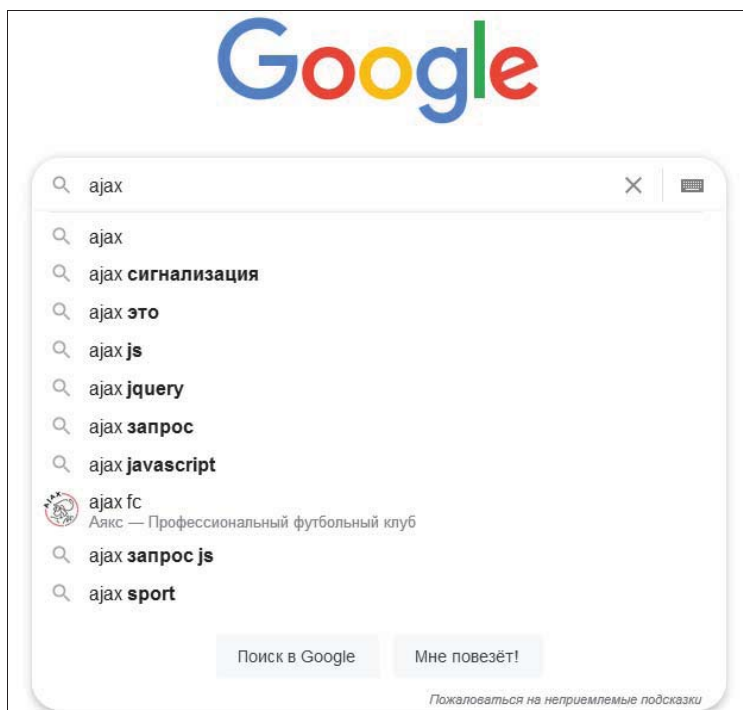


Рис. 25.1. Пример использования технологии AJAX

Для создания фрейма используется тег `<iframe>` с нулевой высотой и шириной. Дополнительно такой фрейм скрывают, указав у него стиль с атрибутом `display`, имеющим значение `none`.

Название этого фрейма указывается в параметре `target` тега `<form>`. Получив данные из формы, серверная программа должна вернуть JavaScript-программу, которая произведет необходимые изменения в текущей странице.

Создадим в корневом каталоге веб-сервера страницу `index.html` (листинг 25.1), которая отправит данные из формы серверной программе `ajax.php` (листинг 25.2).

#### Листинг 25.1. Использование тега `<iframe>`

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <meta name="viewport"
 content="width=device-width, initial-scale=1, shrink-to-fit=no">
 <link rel="stylesheet" type="text/css" href="css/bootstrap.min.css">
 <title>Использование тега <iframe></title>
</head>
<body>
<div class="container my-3">
```

```

<form action="ajax.php" method="GET" target="frame_ajax">
 <div class="form-group">
 <input type="text" class="form-control" name="txt1" id="txt1">
 </div>
 <button type="submit" class="btn btn-primary">Отправить</button>
</form>
</div>
<div class="container my-3">
 <div id="div_ajax"></div>
 <iframe name="frame_ajax" src="ajax.php" width="0"
 height="0" style="display: none"></iframe>
</div>
</body>
</html>

```

Для оформления страницы воспользуемся библиотекой Bootstrap 4.6. В предыдущих главах мы установили эту библиотеку с помощью NPM. Найдите в каталоге `C:\book\p1\node_modules\bootstrap\dist\css` файл `bootstrap.min.css`, содержащий CSS-код этой библиотеки, и поместите в каталог `css`, расположенный в корневом каталоге веб-сервера.

#### Листинг 25.2. Серверная программа `ajax.php`

```

<?php
// Запрещаем кэширование
header('Expires: Thu, 01 Jul 2020 01:00:00 GMT');
header('Cache-Control: no-store, no-cache, must-revalidate');
header('Pragma: no-cache');
// Указываем MIME-тип и кодировку
header('Content-Type: text/html; charset=utf-8');
if (isset($_GET['txt1'])) {
 $msg = 'Данные - ' . $_GET['txt1'];
 echo '<script>', "\n";
 echo 'parent.document.getElementById("txt1").value = "' . $msg . '";';
 echo 'parent.document.getElementById("div_ajax").innerHTML = "' . $msg . '";';
 echo 'htmlspecialchars($msg, ENT_COMPAT, "UTF-8"), "' . $msg . '";';
 echo "\n";
 echo "</script>\n";
}
else {
 echo 'Данные не получены';
}

```

Запустим веб-сервер, откроем веб-браузер и перейдем по интернет-адресу **`http://localhost/`**. Когда страница `index.html` откроется, введем в текстовое поле произвольное значение и щелкнем кнопку **Отправить**. Под формой появится надпись вида "Данные - <Введенное значение>".

Получив введенное в текстовое поле `txt1` значение, серверная программа `ajax.php` создаст переменную `$_GET['txt1']`, хранящую это значение. Далее она сформирует и перешлет странице в составе серверного ответа следующий фрагмент HTML-кода со скриптом

```
<script>
parent.document.getElementById("txt1").value = "";
parent.document.getElementById("div_ajax").innerHTML =
 "Данные - <Введенное значение>";
</script>
```

Этот код загружается в присутствующий на странице фрейм с названием `frame_ajax` и сразу же обрабатывается веб-браузером. Входящий в состав этого кода скрипт выводит строку указанного ранее формата внутри тега `<div>` с идентификатором `div_ajax`.

Для получения ссылки на окно, в котором находится фрейм, используется свойство `parent` объекта `window`. Также можно воспользоваться свойством `top`, которое возвращает ссылку на самое верхнее родительское окно, например:

```
top.document.getElementById("txt1").value = "";
top.document.getElementById("div_ajax").innerHTML =
 "Данные - <Введенное значение>";
```

Чтобы исключить кэширование данных, в самом начале серверной программы мы отправляем целый набор заголовков, запрещающих кэширование:

```
header('Expires: Thu, 01 Jul 2020 01:00:00 GMT');
header('Cache-Control: no-store, no-cache, must-revalidate');
header('Pragma: no-cache');
```

Эти заголовки предназначены для разных веб-браузеров. В заголовке `Expires` мы указываем заведомо прошедшую дату, чтобы веб-браузер, "знакомый" с этим заголовком, не сохранял полученные данные в кэше.

Еще у отправляемого ответа следует указать MIME-тип и кодировку, добавив соответствующий заголовок вызовом функции `header()`:

```
header('Content-Type: text/html; charset=utf-8');
```

Если кодировка в ответе не указана, веб-браузер предполагает, что ответ пришел в кодировке по умолчанию — UTF-8. Поэтому, если данные посылаются в другой кодировке, ее обязательно необходимо задать.

### **ВНИМАНИЕ!**

Так как заголовки должны присутствовать в самом начале серверного ответа, перед вызовом функции `header()` не должно быть никаких операторов вывода. Даже пустая строка перед открывающим PHP-тегом (`<?php`) вызовет ошибку. Пустые строки внутри PHP-тега ошибку не генерируют, поскольку вывод информации осуществляется только с помощью операторов вывода. Кроме того, файл должен быть сохранен в кодировке UTF-8 без метки порядка байтов (BOM), поскольку таковая станет причиной ошибки.

Использование тега `<iframe>` не является единственной альтернативой технологии AJAX. Вместо этого тега можно динамически изменять параметр `src` тега `<script>`, передавая данные в составе интернет-адреса через GET-параметры. В ответ серверная программа также должна сформировать фрагмент HTML-кода, содержащий скрипт, который изменит содержимое текущей страницы без ее перезагрузки.

Однако технология AJAX предоставляет гораздо больше возможностей. Мы можем получить ответ от сервера не только в виде фрагмента HTML-кода, но и в виде объекта JSON, XML-документа, или простого текста. Передать запрос на сервер можно как методом GET, так и методом POST. Рассмотрим технологию AJAX более подробно.

### 25.1.2. Объект *XMLHttpRequest*

Для обмена данными между веб-браузером и серверной программой применяется объект класса *XMLHttpRequest*. Этот объект в фоновом режиме может отправить запрос и получить результат, который обрабатывается программно, в скрипте.

Прежде чем использовать объект *XMLHttpRequest*, необходимо его создать с помощью оператора `new`:

```
var req = new XMLHttpRequest();
```

Объект *XMLHttpRequest* содержит следующие методы:

□ `open(<Метод передачи данных>, <Интернет-адрес>[, <Тип запроса>])` — подготавливает запрос к отправке на сервер. В первом параметре можно указать значения 'GET', 'POST' или др. Во втором параметре указывается интернет-адрес серверной программы, которой будет отправлен запрос (в виде строки или объекта URL). Следует иметь в виду, что серверная программа должна размещаться на том же домене, с которого была загружена текущая страница. Кроме того, некоторые веб-браузеры не позволяют отправить запрос к домену **http://www.site.ru/**, если страница была загружена с **http://site.ru/**. Поэтому лучше указывать сокращенный интернет-адрес с путем от корня сайта или от местонахождения текущей страницы. Если данные отправляются с помощью метода GET, то после имени серверной программы можно указать передаваемые ей GET-параметры. В третьем параметре могут быть заданы следующие значения:

- `true` — асинхронный запрос (выполнение текущего скрипта не приостанавливается до получения ответа; значение по умолчанию);
- `false` — синхронный запрос (выполнение текущего скрипта приостановится и будет продолжено только после получения ответа сервера).

В большинстве случаев встречаются асинхронные запросы, т. к. ответ от сервера может занять некоторое время.

Метод `open()` имеет еще два необязательных параметра: имя пользователя и пароль. На практике эти параметры применяются очень редко, и мы их рассматривать не будем;



- ❑ `setRequestHeader(<Имя>, <Значение>)` — добавляет в отправляемый запрос заголовок с указанными именем и значением. В большинстве случаев используется для задания типа данных, передаваемых методом POST, например:

```
req.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
```

- ❑ `overrideMimeType(<MIME-тип>)` — задает MIME-тип полученного серверного ответа, если сервер его не передал или передал неправильно. Метод поддерживается не всеми веб-браузерами, поэтому лучше перед вызовом проверить его наличие следующим образом:

```
if (req.overrideMimeType) {
 req.overrideMimeType('text/xml');
}
```

- ❑ `getAllResponseHeaders()` — возвращает строку со всеми заголовками серверного ответа (кроме заголовка Set-Cookie) или пустую строку в случае ошибки:

```
if (req.readyState == 4) {
 console.log(req.getAllResponseHeaders());
}
```

Заголовки разделяются двумя символами: `\r\n`. Имя заголовка от его значения отделяется двоеточием и пробелом:

```
Date: Mon, 31 May 2021 14:07:32 GMT
Expires: Tue, 01 June 2021 01:00:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
Content-Length: 24
Content-Type: text/html; charset=utf-8
```

- ❑ `getResponseHeader(<Имя>)` — возвращает значение заголовка серверного ответа с указанным именем или значение `null`, если заголовок не найден или при его извлечении возникла ошибка:

```
if (req.readyState == 4) {
 console.log(req.getResponseHeader('Content-Type'));
 // text/html; charset=utf-8
}
```

- ❑ `send([<Данные для отправки методом POST>])` — отправляет запрос на сервер. В качестве параметра при запросе методом GET указывается значение `null` или вообще ничего:

```
req.send();
```

При отправке данных методом POST указывается строка передаваемых данных. Передаваемые данные должны быть закодированы с помощью функции `encodeURIComponent()`. Пример:

```
var formData = 'txt1=' + encodeURIComponent('Текст');
req.send(formData);
```

- ❑ `abort()` — отменяет текущий запрос.

Свойства объекта XMLHttpRequest:

- `onreadystatechange` — ссылка на функцию-обработчик, обрабатывающую полученные от серверной программы данные при асинхронных запросах:

```
req.onreadystatechange = processReqChange;
```

Так как в функцию-обработчик невозможно передать параметр, то внутри функции мы можем обратиться к объекту XMLHttpRequest, только в том случае, если он будет объявлен в глобальной области видимости, что не всегда удобно. Однако передать параметр можно, указав в качестве обработчика анонимную функцию, внутри которой происходит вызов функции с параметром, например:

```
req.onreadystatechange = function() {
 processReqChange(req);
};
```

Этим способом мы и будем пользоваться в дальнейших примерах.

Функция-обработчик, заданная в свойстве `onreadystatechange`, будет вызываться несколько раз при каждом изменении статуса обработки запроса;

- `readyState` — текущий статус обработки запроса. Может принимать следующие значения:
  - 0 — UNSENT — объект запроса создан оператором `new`;
  - 1 — OPENED — запрос подготовлен с помощью метода `open()`;
  - 2 — HEADERS\_RECEIVED — запрос отправлен методом `send()`, и получены заголовки серверного ответа;
  - 3 — LOADING — ответ доступен, но загружен не полностью;
  - 4 — DONE — ответ полностью загружен с сервера и доступен для обработки, или что-то пошло не так.

В дальнейшем нас будет интересовать статус 4;

- `status` — код статуса ответа при значении свойства `readyState`, равном 4. Успешными считаются коды в пределах от 200 до 299;
- `statusText` — статус ответа в виде строки (например, 'ОК' для кода 200);
- `timeout` — максимальное время ожидания серверного ответа в миллисекундах:  

```
req.timeout = 5000; // 5 секунд
```
- `ontimeout` — ссылка на функцию, которая будет вызвана, если истекло время ожидания, заданное с помощью свойства `timeout`:  

```
req.ontimeout = function() {
 document.getElementById('div_ajax').innerHTML =
 'Истекло время ожидания';
};
```
- `onabort` — ссылка на функцию, которая будет вызвана, если запрос был отменен вызовом метода `abort()`:

```
req.onabort = function() {
 document.getElementById('div_ajax').innerHTML =
 'Вызван метод abort()';
};
```

- **onerror** — ссылка на функцию, которая будет вызвана, если запрос не может быть выполнен (например, отсутствует соединение с Интернетом):

```
req.onerror = function() {
 window.alert('Ошибка');
};
```

- **onloadstart** — ссылка на функцию, которая будет вызвана в начале загрузки серверного ответа:

```
req.onloadstart = function() {
 document.getElementById('div_ajax').innerHTML = 'Загрузка...';
};
```

- **onload** — ссылка на функцию, которая будет вызвана в конце загрузки серверного ответа, включая ответы при ошибках, например при ошибке 404 (страница не найдена):

```
req.onload = function() {
 if (req.status == 200) { // Запрос успешно обработан
 console.log(req.responseText);
 }
 else {
 console.log('Статус: ' + req.status);
 }
};
```

- **onloadend** — ссылка на функцию, которая будет вызвана в конце выполнения запроса вне зависимости от успешности выполнения;

- **onprogress** — ссылка на функцию, которая будет периодически вызываться в процессе загрузки ответа. Внутри функции через параметр будет доступен объект события со следующими свойствами:

- **lengthComputable** — значение `true`, если размер полученного серверного ответа удалось определить (в составе ответа присутствует заголовок `Content-Length`);
- **loaded** — число загруженных байтов;
- **total** — размер серверного ответа в байтах (если его удалось определить).

Пример:

```
req.onprogress = function(e) {
 console.log('lengthComputable : ' + e.lengthComputable);
 console.log('Загружено : ' + e.loaded + ' из ' + e.total);
};
```

- `upload` — ссылка на объект, позволяющий обработать события передачи данных на сервер методом POST (например, события отправки на сервер большого файла). Назначить обработчики событий можно с помощью свойств `onloadstart`, `onprogress`, `onabort`, `onerror`, `onload`, `ontimeout` и `onloadend`. Пример:

```
req.upload.onprogress = function(e) {
 console.log('Отправлено : ' + e.loaded + ' из ' + e.total);
};
req.upload.onloadend = function() {
 console.log('Отправка данных завершена');
};
```

- `withCredentials` — если `true`, другому домену будут отправлены cookie и заголовки HTTP-авторизации, если `false` — не будут отправлены (поведение по умолчанию).

Получить доступ к серверному ответу можно с помощью следующих свойств:

- `responseText` — ответ в виде текста, фрагмента HTML-кода или объекта JSON;
- `responseXML` — ответ в виде XML-документа или значение `null`. Свойство будет доступно, только в том случае, если в заголовках ответа сервера указан заголовок

`Content-Type: application/xml`

или

`Content-Type: text/xml`

Кроме того, XML-документ должен быть иметь правильную структуру и соответствовать спецификации DOM.

Вместо свойств `responseText` и `responseXML` удобнее использовать свойство `response`, но предварительно нужно указать ожидаемый тип серверного ответа с помощью свойства `responseType`:

- `''` (значение по умолчанию) или `'text'` — текст;
- `'document'` — фрагмент HTML-кода или XML-документ;
- `'json'` — данные в формате JSON (будут автоматически преобразованы в объект);
- `'arraybuffer'` — бинарные данные в виде объекта `ArrayBuffer`;
- `'blob'` — бинарные данные в виде объекта `Blob`.

При ошибке свойство `response` может вернуть значение `null`. Пример получения текста:

```
req.responseType = 'text';
req.onload = function() {
 if (req.status == 200) { // Запрос успешно обработан
 var serverData = req.response;
 console.log(serverData);
 }
};
```

### 25.1.3. Получение данных в текстовом формате

Проще всего получить от серверной программы результат в виде HTML-фрагмента, скрипта JavaScript или обычного текста. Серверный ответ будет сохранен в свойствах `responseText` и `response` объекта `XMLHttpRequest`. Для вывода результата достаточно присвоить его свойству `innerHTML` какого-либо существующего элемента страницы.

Рассмотрим получение данных в текстовом формате на примере. Создадим в корневом каталоге веб-сервера страницу `index.html` (листинг 25.3), которая будет отправлять данные из формы серверной программе `ajax.php` (листинг 25.4) HTTP-методами GET и POST.

#### Листинг 25.3. Получение данных в текстовом формате

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <meta name="viewport"
 content="width=device-width, initial-scale=1, shrink-to-fit=no">
 <link rel="stylesheet" type="text/css" href="css/bootstrap.min.css">
 <title>Получение данных в текстовом формате</title>
</head>
<body>
<div class="container my-3">
 <form action="ajax.php" method="GET" onsubmit="return false;">
 <div class="form-group">
 <input type="text" class="form-control" id="txt1">
 </div>
 <button type="button" class="btn btn-primary"
 id="btnSendGET">Отправить методом GET</button>
 <button type="button" class="btn btn-primary"
 id="btnSendPOST">Отправить методом POST</button>
 </form>
</div>
<div class="container my-3">
 <div id="div_ajax"></div>
</div>

<script>
function processReqChange(req) {
 // Обрабатываем асинхронный запрос
 try {
 if (req.readyState == 4) {
 if (req.status == 200) { // Запрос успешно обработан
 var serverData = req.responseText;
```

```
 document.getElementById('txt1').value = '';
 document.getElementById('div_ajax').innerHTML = serverData;
 }
 else {
 document.getElementById('div_ajax').innerHTML = 'Ошибка';
 }
}
}
catch(e) {
 document.getElementById('div_ajax').innerHTML = 'Ошибка';
}
}

function sendReqGET() {
 // Отправка запроса методом GET
 if (!window.XMLHttpRequest) {
 window.alert('Ваш веб-браузер не поддерживает XMLHttpRequest');
 return;
 }
 var txt = document.getElementById('txt1').value;
 if (txt === '') {
 window.alert('Поле не заполнено');
 return;
 }
 var url = '/ajax.php?txt1=' + encodeURIComponent(txt);
 // Создание объекта XMLHttpRequest
 var req = new XMLHttpRequest();
 // Подготовка асинхронного запроса методом GET
 req.open('GET', url, true);
 // Обработка событий
 req.onreadystatechange = function() {
 processReqChange(req);
 };
 req.onloadstart = function() {
 document.getElementById('div_ajax').innerHTML = 'Загрузка...';
 };
 // Отправка запроса
 req.send();
}

function sendReqPOST() {
 // Отправка запроса методом POST
 if (!window.XMLHttpRequest) {
 window.alert('Ваш веб-браузер не поддерживает XMLHttpRequest');
 return;
 }
}
```

```

var txt = document.getElementById('txt1').value;
if (txt === '') {
 window.alert('Поле не заполнено');
 return;
}
// Передаваемые параметры
var formData = 'txt1=' + encodeURIComponent(txt);
// Создание объекта XMLHttpRequest
var req = new XMLHttpRequest();
// Подготовка асинхронного запроса методом POST
req.open('POST', '/ajax.php', true);
// Добавляем HTTP-заголовок
req.setRequestHeader('Content-Type',
 'application/x-www-form-urlencoded');
// Обработка событий
req.onreadystatechange = function() {
 processReqChange(req);
};
req.onloadstart = function() {
 document.getElementById('div_ajax').innerHTML = 'Загрузка...';
};
// Отправка запроса
req.send(formData);
}
document.getElementById('btnSendGET').onclick = sendReqGET;
document.getElementById('btnSendPOST').onclick = sendReqPOST;
</script>
</body>
</html>

```

#### Листинг 25.4. Серверная программа ajax.php

```

<?php
// Запрещаем кэширование
header('Expires: Thu, 01 Jul 2020 01:00:00 GMT');
header('Cache-Control: no-store, no-cache, must-revalidate');
header('Pragma: no-cache');
// Указываем MIME-тип и кодировку
header('Content-Type: text/html; charset=utf-8');
if (isset($_GET['txt1'])) {
 echo 'Метод GET - ';
 echo htmlspecialchars($_GET['txt1'], ENT_COMPAT, 'UTF-8');
}
else {
 if (isset($_POST['txt1'])) {
 echo 'Метод POST - ';
 echo htmlspecialchars($_POST['txt1'], ENT_COMPAT, 'UTF-8');
 }
}

```

```
else {
 echo 'Данные не получены';
}
}
```

Страница, код которой приведен в листинге 25.3, содержит текстовое поле и две кнопки. При щелчке на кнопке **Отправить методом GET** будет вызвана функция `sendReqGET()`, а при щелчке на кнопке **Отправить методом POST** — функция `sendReqPOST()`. Результат выполнения запроса появится в теге `<div>` с идентификатором `div_ajax`.

Внутри функции `sendReqGET()` сначала проверяем, поддерживает ли веб-браузер класс `XMLHttpRequest`. Если не поддерживает, выводим сообщение и выходим из функции:

```
if (!window.XMLHttpRequest) {
 window.alert('Ваш веб-браузер не поддерживает XMLHttpRequest');
 return;
}
```

В противном случае получаем значение из текстового поля и сохраняем его в переменной `txt`:

```
var txt = document.getElementById('txt1').value;
```

Проверяем, заполнено ли текстовое поле, и если оно не заполнено, выводим сообщение и завершаем выполнение функции:

```
if (txt === '') {
 window.alert('Поле не заполнено');
 return;
}
```

Формируем интернет-адрес с GET-параметром, содержащим введенное в форму значение, которое кодируем с помощью функции `encodeURIComponent()`. Сформированный адрес сохраняем в переменной `url`:

```
var url = '/ajax.php?txt1=' + encodeURIComponent(txt);
```

Создаем объект `XMLHttpRequest`:

```
var req = new XMLHttpRequest();
```

С помощью метода `open()` подготавливаем объект к отправке асинхронного запроса методом GET:

```
req.open('GET', url, true);
```

В свойстве `onreadystatechange` указываем анонимную функцию, в которой вызывается функция `processReqChange()`. Благодаря этому можем передать объект в качестве параметра функции:

```
req.onreadystatechange = function() {
 processReqChange(req);
};
```



С помощью метода `send()` отправляем запрос на сервер. Так как мы передаем данные методом GET, вызываем метод `send()` без параметров:

```
req.send();
```

Поскольку поступление ответа от сервера может произойти через некоторое (иногда продолжительное) время, выводим в элементе с идентификатором `div_ajax` сообщение о том, что идет загрузка данных:

```
req.onloadstart = function() {
 document.getElementById('div_ajax').innerHTML = 'Загрузка...';
};
```

Функция `processReqChange()` будет вызываться при каждом изменении статуса запроса. Внутри этой функции проверяем значение свойства `readyState`. Если оно равно 4, то ответ полностью получен с сервера и доступен для обработки:

```
if (req.readyState == 4) {
 . . .
}
```

Наличие статуса завершения еще не гарантирует успешность выполнения запроса. Поэтому мы также проверяем значение свойства `status`. Если оно равно 200, то запрос выполнен успешно. В этом случае получаем ответ из свойства `responseText`, очищаем текстовое поле и выводим полученное значение в элементе с идентификатором `div_ajax`:

```
if (req.status == 200) {
 var serverData = req.responseText;
 document.getElementById('txt1').value = '';
 document.getElementById('div_ajax').innerHTML = serverData;
}
```

Чтобы исключить различные ошибки, все тело функции `processReqChange()` расположено внутри конструкции `try...catch`.

Внутри функции `sendReqPOST()` мы также получаем значение текстового поля, сохраняем его в переменной `txt`, проверяем, заполнено ли текстовое поле, и если оно не заполнено, выводим сообщение и завершаем выполнение функции. Затем формируем строку с данными, которую будем передавать методом POST. Значение, введенное пользователем в текстовое поле, кодируем с помощью функции `encodeURIComponent()`. Сформированную строку сохраняем в переменной `formData`:

```
var formData = 'txt1=' + encodeURIComponent(txt);
```

Создаем объект `XMLHttpRequest` и с помощью метода `open()` подготавливаем его к отправке асинхронного запроса методом POST:

```
req.open('POST', '/ajax.php', true);
```

При передаче данных методом POST необходимо указать, что данные переданы из формы, добавив в запрос заголовок `Content-Type` со значением `application/x-www-form-urlencoded`. Это можно сделать, вызвав метод `setRequestHeader()`:

```
req.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
```

Далее, как и ранее, в свойстве `onreadystatechange` указываем функцию-обработчик. Затем с помощью метода `send()` отправляем запрос на сервер, в качестве параметра указав сформированную строку с данными из переменной `formData`:

```
req.send(formData);
```

Дальнейшая обработка запроса происходит точно так же, как и при отправке запроса методом GET.

Получив клиентский запрос, серверная программа `ajax.php` создаст следующие переменные окружения:

□ при передаче методом GET:

```
$_GET['txt1']
```

□ при передаче методом POST:

```
$_POST['txt1']
```

Проверить существование переменных можно с помощью функции `isset()`. Далее просто возвращаем отправленное значение обратно веб-браузеру, дополнительно указав, каким HTTP-методом были получены данные.

## 25.1.4. Получение данных в формате XML

Прежде чем рассматривать обработку данных в формате XML с помощью JavaScript, напомним правила создания XML-документов.

XML-документ состоит из двух разделов — пролога и тела документа. В прологе указываются используемая версия языка XML и кодировка документа:

```
<?xml version="1.0" encoding="utf-8" ?>
```

Если кодировка не указана, по умолчанию будет принята UTF-8.

Тело XML-документа должно находиться внутри корневого тега:

```
<?xml version="1.0" encoding="utf-8" ?>
<ajax>
<!-- Тело XML-документа -->
</ajax>
```

Теги XML-документа могут иметь произвольные имена:

```
<ajax>
 <item id="1">
 <name>HTML</name>
 </item>
 <item id="2">
 <name>CSS</name>
 <version>3</version>
 </item>
</ajax>
```

Только следует учитывать, что имена тегов должны начинаться с буквы и содержать буквы, цифры или символ подчеркивания. К тому же регистр букв, которыми набираются имена тегов, имеет значение.

Каждому открывающему тегу обязательно должен соответствовать закрывающий:

```
<name>HTML</name>
```

XML-теги могут иметь параметры. Параметр указывается в формате:

```
<Имя параметра>="<Значение>"
```

Значение параметра обязательно должно быть указано в кавычках, а сам параметр ставится в открывающем теге:

```
<item id="2">. . .</item>
```

Все специальные символы внутри тегов и в значениях параметров должны быть заменены следующими последовательностями:

- ☐ `&lt;` — знак меньше (<);
- ☐ `&gt;` — знак больше (>);
- ☐ `&amp;` — амперсанд (&);
- ☐ `&quot;` — кавычка (");
- ☐ `&apos;` — апостроф (').

Внутри тегов можно записывать комментарии, синтаксис которых похож на синтаксис комментариев HTML:

```
<!-- Это комментарий -->
```

Теперь рассмотрим получение данных в формате XML на примере. Создадим в корневом каталоге веб-сервера страницу `index.html` (листинг 25.5), которая отправит данные из формы серверной программе `ajax.php` (листинг 25.6).

#### Листинг 25.5. Получение данных в формате XML

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <meta name="viewport"
 content="width=device-width, initial-scale=1, shrink-to-fit=no">
 <link rel="stylesheet" type="text/css" href="css/bootstrap.min.css">
 <title>Получение данных в формате XML</title>
</head>
<body>
<div class="container my-3">
 <form action="ajax.php" method="GET" onsubmit="return false;">
 <div class="form-group">
 <input type="text" class="form-control" id="txt1">
 </div>
```

```
<div class="form-group">
 <input type="text" class="form-control" id="txt2">
</div>
<button type="button" class="btn btn-primary"
 id="btnSend">Отправить методом GET</button>
</form>
</div>
<div class="container my-3">
 <div id="div_ajax"></div>
</div>

<script>
function processReqChange(req) {
 // Обрабатываем асинхронный запрос
 try {
 if (req.readyState == 4) {
 if (req.status == 200) { // Запрос успешно обработан
 var xml = req.responseXML;
 var serverData = xml.getElementsByTagName('serverData');
 var txt1 = serverData[0].getElementsByTagName('txt1')[0];
 var txt2 = serverData[0].getElementsByTagName('txt2')[0];
 var msg = txt1.firstChild.data + '
' +
 txt2.firstChild.data;
 document.getElementById('txt1').value = '';
 document.getElementById('txt2').value = '';
 document.getElementById('div_ajax').innerHTML = msg;
 }
 else {
 document.getElementById('div_ajax').innerHTML = 'Ошибка';
 }
 }
 }
 catch(e) {
 document.getElementById('div_ajax').innerHTML = 'Ошибка';
 }
}

function sendReqGET() {
 // Отправка запроса методом GET
 if (!window.XMLHttpRequest) {
 window.alert('Ваш веб-браузер не поддерживает XMLHttpRequest');
 return;
 }
 var txt1 = document.getElementById('txt1').value;
 var txt2 = document.getElementById('txt2').value;
 if (txt1 === '' || txt2 === '') {
 window.alert('Поле не заполнено');
```

```

 return;
 }
 var url = '/ajax.php?txt1=' + encodeURIComponent(txt1);
 url += '&txt2=' + encodeURIComponent(txt2);
 // Создание объекта XMLHttpRequest
 var req = new XMLHttpRequest();
 // Подготовка асинхронного запроса методом GET
 req.open('GET', url, true);
 req.onreadystatechange = function() {
 processReqChange(req);
 };
 req.onloadstart = function() {
 document.getElementById('div_ajax').innerHTML = 'Загрузка...';
 };
 // Отправка запроса
 req.send();
}
document.getElementById('btnSend').onclick = sendReqGET;
</script>
</body>
</html>

```

#### Листинг 25.6. Серверная программа ajax.php

```

<?php
// Запрещаем кэширование
header('Expires: Thu, 01 Jul 2020 01:00:00 GMT');
header('Cache-Control: no-store, no-cache, must-revalidate');
header('Pragma: no-cache');
// Указываем MIME-тип и кодировку
header('Content-Type: text/xml; charset=utf-8');
echo "<?xml version='1.0' encoding='utf-8' ?>", "\n";
echo "<ajax>\n";
echo " <serverData>\n";
if (isset($_GET['txt1']) && isset($_GET['txt2'])) {
 echo " <txt1>";
 echo "Поле 1 - ";
 echo htmlspecialchars($_GET['txt1'], ENT_COMPAT, 'UTF-8');
 echo "</txt1>\n";
 echo " <txt2>";
 echo "Поле 2 - ";
 echo htmlspecialchars($_GET['txt2'], ENT_COMPAT, 'UTF-8');
 echo "</txt2>\n";
}
echo " </serverData>\n";
echo "</ajax>\n";

```

После заполнения обоих текстовых полей и щелчка на кнопке **Отправить методом GET** будет вызвана функция `sendReqGET()`. Внутри нее получаем введенные значения, сохраняем их в переменных `txt1` и `txt2`, проверяем, были ли введены значения, и если нет, выводим сообщение и завершаем выполнение функции.

Затем формируем интернет-адрес с GET-параметрами, содержащими введенные значения, не забыв закодировать их с помощью функции `encodeURIComponent()`. Сформированный интернет-адрес сохраняем в переменной `url`:

```
var url = '/ajax.php?txt1=' + encodeURIComponent(txt1);
url += '&txt2=' + encodeURIComponent(txt2);
```

Создаем объект `XMLHttpRequest` и с помощью метода `open()` подготавливаем его к отправке асинхронного запроса методом GET:

```
req.open('GET', url, true);
```

В свойстве `onreadystatechange` указываем функцию-обработчик и с помощью метода `send()` отправляем запрос на сервер:

```
req.send();
```

Получив запрос, серверная программа создаст переменные окружения `$_GET['txt1']` и `$_GET['txt2']`.

Для корректного отображения отправленного ответа указываем MIME-тип и кодировку с помощью функции `header()`:

```
header('Content-Type: text/xml; charset=utf-8');
```

Если MIME-тип не указан, получить данные из свойства `responseXML` не получится. Однако можно после создания объекта `XMLHttpRequest` до вызова метода `send()` явным образом указать MIME-тип с помощью метода `overrideMimeType()`, не полагаясь на заголовки серверного ответа:

```
if (req.overrideMimeType) {
 req.overrideMimeType('text/xml');
}
```

Далее формируем XML-документ, который отправим в составе ответа. В результате веб-браузер получит следующий XML-документ:

```
<?xml version="1.0" encoding="utf-8" ?>
<ajax>
 <serverData>
 <txt1>Поле 1 - <Значение из текстового поля 1></txt1>
 <txt2>Поле 2 - <Значение из текстового поля 2></txt2>
 </serverData>
</ajax>
```

После получения ответа XML-документ будет доступен через свойство `responseXML` внутри функции `processReqChange()`:

```
var xml = req.responseXML;
```

С помощью метода `getElementsByTagName()` получаем коллекцию всех элементов `serverData`, содержащихся в полученном XML-документе, и сохраняем ее в переменной `serverData`:

```
var serverData = xml.getElementsByTagName('serverData');
```

Такой элемент в полученном документе всего один, и мы далее извлечем его, указав его индекс, равный 0, в квадратных скобках.

Элементы `txt1` и `txt2`, которые находятся внутри элемента `serverData` в XML-документе, также получаем с помощью метода `getElementsByTagName()`. Каждый из этих элементов также всего один, и мы извлекаем его точно таким же способом:

```
var txt1 = serverData[0].getElementsByTagName('txt1')[0];
var txt2 = serverData[0].getElementsByTagName('txt2')[0];
```

Чтобы получить данные внутри элементов `txt1` и `txt2`, следует воспользоваться свойством `firstChild`. Свойство `firstChild` содержит информацию о длине строки (свойство `length`) и текстовых данных элемента (свойство `data`). Именно свойство `data` позволяет нам получить переданные данные:

```
var msg = txt1.firstChild.data + '
' + txt2.firstChild.data;
document.getElementById('div_ajax').innerHTML = msg;
```

В нашем примере всего один элемент `serverData`, поэтому достаточно сразу найти элементы `txt1` и `txt2`. Рассмотрим это на примере (листинг 25.7), а заодно укажем тип документа в свойстве `responseType` и получим серверный ответ с помощью свойства `response`.

#### Листинг 25.7. Свойства `responseType` и `response`

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <meta name="viewport"
 content="width=device-width, initial-scale=1, shrink-to-fit=no">
 <link rel="stylesheet" type="text/css" href="css/bootstrap.min.css">
 <title>Свойства responseType и response</title>
</head>
<body>
<div class="container my-3">
 <form action="ajax.php" method="GET" onsubmit="return false;">
 <div class="form-group">
 <input type="text" class="form-control" id="txt1">
 </div>
 <div class="form-group">
 <input type="text" class="form-control" id="txt2">
 </div>
```

```
<button type="button" class="btn btn-primary"
 id="btnSend">Отправить методом GET</button>
</form>
</div>
<div class="container my-3">
 <div id="div_ajax"></div>
</div>

<script>
function onloadXML(req) {
 try {
 if (req.status == 200) { // Запрос успешно обработан
 var xml = req.response;
 var txt1 = xml.getElementsByTagName('txt1')[0];
 var txt2 = xml.getElementsByTagName('txt2')[0];
 var msg = txt1.firstChild.data + '
' +
 txt2.firstChild.data;
 document.getElementById('txt1').value = '';
 document.getElementById('txt2').value = '';
 document.getElementById('div_ajax').innerHTML = msg;
 }
 else {
 document.getElementById('div_ajax').innerHTML = 'Ошибка';
 }
 }
 catch(e) {
 document.getElementById('div_ajax').innerHTML = 'Ошибка';
 }
}

function sendReqGET() {
 // Отправка запроса методом GET
 if (!window.XMLHttpRequest) {
 window.alert('Ваш веб-браузер не поддерживает XMLHttpRequest');
 return;
 }
 var txt1 = document.getElementById('txt1').value;
 var txt2 = document.getElementById('txt2').value;
 if (txt1 === '' || txt2 === '') {
 window.alert('Поле не заполнено');
 return;
 }
 var url = '/ajax.php?txt1=' + encodeURIComponent(txt1);
 url += '&txt2=' + encodeURIComponent(txt2);
 // Создание объекта XMLHttpRequest
 var req = new XMLHttpRequest();
```



```
// Подготовка асинхронного запроса методом GET
req.open('GET', url, true);
req.responseType = 'document';
req.onerror = function() {
 document.getElementById('div_ajax').innerHTML = 'Ошибка';
};
req.onload = function() {
 onloadXML(req);
};
req.onloadstart = function() {
 document.getElementById('div_ajax').innerHTML = 'Загрузка...';
};
// Отправка запроса
req.send(null);
}
document.getElementById('btnSend').onclick = sendReqGET;
</script>
</body>
</html>
```

В этом примере мы указали тип документа в свойстве `responseType`:

```
req.responseType = 'document';
```

Даже если серверная программа не отправит заголовок с MIME-типом, ответ все равно будет правильно обработан, а содержащийся в нем XML-документ будет доступен через свойство `response`:

```
var xml = req.response;
```

### 25.1.5. Получение данных в формате JSON

Данные от серверной программы удобнее получать в формате *JSON* (JavaScript Object Notation, нотация объекта JavaScript). XML-документ из предыдущего примера можно представить в формате JSON следующим образом:

```
var json = '{' +
 '"txt1": "Поле 1 - Строка 1", ' +
 '"txt2": "Поле 2 - Строка 2"' +
 '}';
```

Для преобразования строки с JSON-данными в объект можно использовать метод `parse()` класса `JSON` (см. *разд. 11.11*) или функцию `eval()`:

```
var jsonObject = null;
try {
 if (window.JSON && window.JSON.parse) {
 jsonObject = JSON.parse(json);
 }
```

```
else {
 jsonObject = eval('(' + json + ')');
}
console.log(jsonObject.txt1); // Поле 1 - Строка 1
console.log(jsonObject.txt2); // Поле 2 - Строка 2
} catch (e) {
 console.log('Не удалось декодировать');
}
```

Рассмотрим пример получения данных в формате JSON. Создадим в корневом каталоге веб-сервера страницу `index.html` (листинг 25.8), которая будет отправлять данные из формы серверной программе `ajax.php` (листинг 25.9).

#### Листинг 25.8. Получение данных в формате JSON

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <meta name="viewport"
 content="width=device-width, initial-scale=1, shrink-to-fit=no">
 <link rel="stylesheet" type="text/css" href="css/bootstrap.min.css">
 <title>Получение данных в формате JSON</title>
</head>
<body>
<div class="container my-3">
 <form action="ajax.php" method="GET" onsubmit="return false;">
 <div class="form-group">
 <input type="text" class="form-control" id="txt1">
 </div>
 <div class="form-group">
 <input type="text" class="form-control" id="txt2">
 </div>
 <button type="button" class="btn btn-primary"
 id="btnSend">Отправить методом GET</button>
 </form>
</div>
<div class="container my-3">
 <div id="div_ajax"></div>
</div>

<script>
function processReqChange(req) {
 // Обрабатываем асинхронный запрос
 try {
 if (req.readyState == 4) {
 if (req.status == 200) { // Запрос успешно обработан
 var json = req.responseText;
```

```

 var serverData = null;
 if (window.JSON && window.JSON.parse) {
 serverData = JSON.parse(json);
 }
 else {
 serverData = eval('(' + json + ')');
 }
 var msg = serverData.txt1 + '
' + serverData.txt2;
 document.getElementById('txt1').value = '';
 document.getElementById('txt2').value = '';
 document.getElementById('div_ajax').innerHTML = msg;
 }
 else {
 document.getElementById('div_ajax').innerHTML = 'Ошибка';
 }
}
}
catch(e) {
 document.getElementById('div_ajax').innerHTML = 'Ошибка';
}
}

function sendReqGET() {
 // Отправка запроса методом GET
 if (!window.XMLHttpRequest) {
 window.alert('Ваш веб-браузер не поддерживает XMLHttpRequest');
 return;
 }
 var txt1 = document.getElementById('txt1').value;
 var txt2 = document.getElementById('txt2').value;
 if (txt1 === '' || txt2 === '') {
 window.alert('Поле не заполнено');
 return;
 }
 var url = '/ajax.php?txt1=' + encodeURIComponent(txt1);
 url += '&txt2=' + encodeURIComponent(txt2);
 // Создание объекта XMLHttpRequest
 var req = new XMLHttpRequest();
 // Подготовка асинхронного запроса методом GET
 req.open('GET', url, true);
 req.onreadystatechange = function() {
 processReqChange(req);
 };
 req.onloadstart = function() {
 document.getElementById('div_ajax').innerHTML = 'Загрузка...';
 };
};

```

```
req.send(null); // Отправка запроса
}
document.getElementById('btnSend').onclick = sendReqGET;
</script>
</body>
</html>
```

**Листинг 25.9. Серверная программа ajax.php**

```
<?php
// Запрещаем кэширование
header('Expires: Thu, 01 Jul 2020 01:00:00 GMT');
header('Cache-Control: no-store, no-cache, must-revalidate');
header('Pragma: no-cache');
// Указываем MIME-тип и кодировку
header('Content-Type: application/json; charset=utf-8');
if (isset($_GET['txt1']) && isset($_GET['txt2'])) {
 $txt1 = htmlspecialchars($_GET['txt1'], ENT_COMPAT, 'UTF-8');
 $txt2 = htmlspecialchars($_GET['txt2'], ENT_COMPAT, 'UTF-8');
 $arr = array('txt1' => $txt1, 'txt2' => $txt2);
 echo json_encode($arr);
}
else {
 $arr = array('txt1' => 'Данные не получены', 'txt2' => '');
 echo json_encode($arr);
}
```

Для работы с JSON-объектами в PHP существуют две функции:

- ☐ `json_encode(<Ассоциативный массив>)` — преобразует заданный ассоциативный массив в JSON-объект и возвращает его;
- ☐ `json_decode(<JSON-объект>[, <Тип результата>])` — выполняет разбор указанного JSON-объекта. Необязательный параметр `<Тип результата>` может принимать следующие значения:
  - `true` — вернуть результат в виде ассоциативного массива;
  - `false` — вернуть результат в виде объекта (значение по умолчанию).

Пример:

```
<?php
$arr = array();
$arr['txt1'] = 'Строка 1';
$arr['txt2'] = 'Строка 2';
$json = json_encode($arr);
$arr2 = json_decode($json);
echo $arr2->txt1, '
'; // Строка 1
echo $arr2->txt2, '
'; // Строка 2
$arr3 = json_decode($json, true);
```

```
echo $arr3['txt1'], '
'; // Строка 1
echo $arr3['txt2']; // Строка 2
```

Укажем тип документа в свойстве `responseType` и получим серверный ответ из свойства `response` (листинг 25.10).

**Листинг 25.10. Свойства `responseType` и `response`**

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <meta name="viewport"
 content="width=device-width, initial-scale=1, shrink-to-fit=no">
 <link rel="stylesheet" type="text/css" href="css/bootstrap.min.css">
 <title>Свойства responseType и response</title>
</head>
<body>
<div class="container my-3">
 <form action="ajax.php" method="GET" onsubmit="return false;">
 <div class="form-group">
 <input type="text" class="form-control" id="txt1">
 </div>
 <div class="form-group">
 <input type="text" class="form-control" id="txt2">
 </div>
 <button type="button" class="btn btn-primary"
 id="btnSend">Отправить методом GET</button>
 </form>
</div>
<div class="container my-3">
 <div id="div_ajax"></div>
</div>

<script>
function onloadJSON(req) {
 try {
 if (req.status == 200) { // Запрос успешно обработан
 var json = req.response;
 var msg = json.txt1 + '
' + json.txt2;
 document.getElementById('txt1').value = '';
 document.getElementById('txt2').value = '';
 document.getElementById('div_ajax').innerHTML = msg;
 }
 else {
 document.getElementById('div_ajax').innerHTML = 'Ошибка';
 }
 }
}
```

```
 catch(e) {
 document.getElementById('div_ajax').innerHTML = 'Ошибка';
 }
}

function sendReqGET() {
 // Отправка запроса методом GET
 if (!window.XMLHttpRequest) {
 window.alert('Ваш веб-браузер не поддерживает XMLHttpRequest');
 return;
 }
 var txt1 = document.getElementById('txt1').value;
 var txt2 = document.getElementById('txt2').value;
 if (txt1 === '' || txt2 === '') {
 window.alert('Поле не заполнено');
 return;
 }
 var url = '/ajax.php?txt1=' + encodeURIComponent(txt1);
 url += '&txt2=' + encodeURIComponent(txt2);
 // Создание объекта XMLHttpRequest
 var req = new XMLHttpRequest();
 // Подготовка асинхронного запроса методом GET
 req.open('GET', url, true);
 req.responseType = 'json';
 req.onerror = function() {
 document.getElementById('div_ajax').innerHTML = 'Ошибка';
 };
 req.onload = function() {
 onloadJSON(req);
 };
 req.onloadstart = function() {
 document.getElementById('div_ajax').innerHTML = 'Загрузка...';
 };
 req.send(null); // Отправка запроса
}

document.getElementById('btnSend').onclick = sendReqGET;
</script>
</body>
</html>
```

В этом примере мы указали тип документа в свойстве `responseType`:

```
req.responseType = 'json';
```

Полученные данные в виде готового JSON-объекта доступны из свойства `response`:

```
var json = req.response;
```

## 25.2. Fetch API

Современные веб-браузеры позволяют выполнять AJAX-запросы не только с помощью объекта `XMLHttpRequest`, но и посредством *Fetch API*. Проверить возможность использования Fetch API можно следующим образом:

```
if (window.fetch) {
 console.log('Fetch API поддерживается');
}
else {
 console.log('Ваш браузер не поддерживает Fetch API');
 // Здесь можно попробовать отправить данные с помощью XMLHttpRequest
}
```

### 25.2.1. Функция *fetch()*

Для выполнения AJAX-запроса предназначена функция `fetch()`. Формат функции:

```
Promise<Response> = fetch(<Интернет-адрес или запрос>[, <Настройки>])
```

В первом параметре указывается интернет-адрес (в виде строки или объекта `URL`) или объект класса `Request`, представляющий клиентский запрос. Во втором параметре можно передать объект с настройками (см. *разд. 25.2.3*). Функция возвращает объект `Promise`, содержащий в себе объект класса `Response`, который представляет полученный серверный ответ. Отправка запроса выполняется асинхронно, по умолчанию методом `GET`, сразу после вызова функции `fetch()`. Обработать получение данных позволяют методы `then()` и `catch()` объекта `Promise` (см. *разд. 14.1*).

В качестве примера переделаем код функции `sendReqGET()` из листинга 25.10 и отправим запрос, используя Fetch API (листинг 25.11). Функция будет получать данные в формате `JSON`, которые отправляются серверной программой `ajax.php` (см. листинг 25.9).

#### Листинг 25.11. Получение данных в формате JSON с помощью Fetch API

```
function sendReqGET() {
 if (!window.fetch) {
 window.alert('Ваш браузер не поддерживает Fetch API');
 return;
 }
 var txt1 = document.getElementById('txt1').value;
 var txt2 = document.getElementById('txt2').value;
 if (txt1 === '' || txt2 === '') {
 window.alert('Поле не заполнено');
 return;
 }
 var url = '/ajax.php?txt1=' + encodeURIComponent(txt1);
 url += '&txt2=' + encodeURIComponent(txt2);
```

```
document.getElementById('div_ajax').innerHTML = 'Загрузка...';
fetch(url)
 .then(function(response) {
 if (!response.ok) {
 throw new Error('Статус: ' + response.status);
 }
 return response.json();
 })
 .then(function(json) {
 var msg = json.txt1 + '
' + json.txt2;
 document.getElementById('txt1').value = '';
 document.getElementById('txt2').value = '';
 document.getElementById('div_ajax').innerHTML = msg;
 })
 .catch(function(e) {
 document.getElementById('div_ajax').innerHTML = 'Ошибка: ' + e;
 })
 .then(function() {
 console.log('Функция выполнится в любом случае');
 });
}
```

Благодаря тому, что функция `fetch()` и методы `then()` и `catch()` возвращают объекты `Promise`, мы можем строить цепочки из вызовов. В функции обратного вызова, зарегистрированной первым методом `then()`, через параметр доступен объект серверного ответа `Response`. Вызов функции осуществляется в любом случае, поэтому мы проверяем значение свойства `ok` объекта `Response`. Свойство `ok` хранит `true`, если HTTP-код находится в пределах от 200 до 299. Если это не так, генерируем исключение `Error`, указывая в текстовом описании код статуса из свойства `status`. Исключение будет перехвачено и доступно внутри функции, зарегистрированной с помощью метода `catch()`.

Для преобразования строки в формате JSON, возвращенной сервером, в объект, предназначен метод `json()` объекта `Response`. Значение, возвращенное методом `json()`, мы возвращаем из функции. Это значение будет доступно внутри функции обратного вызова, зарегистрированной вторым методом `then()`, через параметр. Если вызов метода `json()` окончился неудачно, генерируется исключение. Исключение будет перехвачено и доступно внутри функции, зарегистрированной с помощью метода `catch()`. Объект исключения доступен внутри функции обратного вызова через параметр.

Функция обратного вызова, зарегистрированная последним методом `then()`, будет вызвана в любом случае, вне зависимости от возникновения исключения, после окончания обработки ответа. Это аналог метода `finally()`.

Веб-браузеры, поддерживающие Fetch API, чаще всего поддерживают ключевые слова `let`, `const` и стрелочные функции, которые удобнее, чем анонимные функции:



```

let url = '/ajax.php?txt1=' + encodeURIComponent(txt1);
url += '&txt2=' + encodeURIComponent(txt2);
document.getElementById('div_ajax').innerHTML = 'Загрузка...';
fetch(url)
 .then(response => {
 if (!response.ok) {
 throw new Error('Статус: ' + response.status);
 }
 return response.json();
 })
 .then(json => {
 const msg = json.txt1 + '
' + json.txt2;
 document.getElementById('txt1').value = '';
 document.getElementById('txt2').value = '';
 document.getElementById('div_ajax').innerHTML = msg;
 })
 .catch(e => {
 document.getElementById('div_ajax').innerHTML = 'Ошибка: ' + e;
 })
 .then(() => console.log('Функция выполнится в любом случае'));

```

Переделаем код функции `sendReqGET()` из листинга 25.11, используя ключевые слова `async` и `await` вместо методов `then()` и `catch()` (листинг 25.12).

#### Листинг 25.12. Ключевые слова `async` и `await`

```

<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <meta name="viewport"
 content="width=device-width, initial-scale=1, shrink-to-fit=no">
 <link rel="stylesheet" type="text/css" href="css/bootstrap.min.css">
 <title>Ключевые слова async и await</title>
</head>
<body>
<div class="container my-3">
 <form action="ajax.php" method="GET" onsubmit="return false;">
 <div class="form-group">
 <input type="text" class="form-control" id="txt1">
 </div>
 <div class="form-group">
 <input type="text" class="form-control" id="txt2">
 </div>
 <button type="button" class="btn btn-primary"
 id="btnSend">Отправить методом GET</button>
 </form>
</div>

```

```
<div class="container my-3">
 <div id="div_ajax"></div>
</div>

<script>
async function sendReqGET() {
 if (!window.fetch) {
 window.alert('Ваш браузер не поддерживает Fetch API');
 return;
 }
 let txt1 = document.getElementById('txt1').value;
 let txt2 = document.getElementById('txt2').value;
 if (txt1 === '' || txt2 === '') {
 window.alert('Поле не заполнено');
 return;
 }
 let url = '/ajax.php?txt1=' + encodeURIComponent(txt1);
 url += '&txt2=' + encodeURIComponent(txt2);
 document.getElementById('div_ajax').innerHTML = 'Загрузка...';
 try {
 let response = await fetch(url);
 if (!response.ok) {
 throw new Error('Статус: ' + response.status);
 }
 let json = await response.json();
 const msg = json.txt1 + '
' + json.txt2;
 document.getElementById('txt1').value = '';
 document.getElementById('txt2').value = '';
 document.getElementById('div_ajax').innerHTML = msg;
 } catch(e) {
 document.getElementById('div_ajax').innerHTML = 'Ошибка: ' + e;
 }
}

document.getElementById('btnSend').onclick = sendReqGET;
</script>
</body>
</html>
```

## 25.2.2. Классы *URL* и *URLSearchParams*

Класс `URL` представляет собой интернет-адрес. Создать экземпляр класса позволяет следующий конструктор:

```
<Объект> = new URL(<URL-адрес>[, <Base>])
```

В первом параметре указывается полный или сокращенный (представляющий собой лишь путь) интернет-адрес в виде строки:

```
let url = new URL('http://localhost/index.php');
console.log(url.toString());
// http://localhost/index.php
```

Если в первом параметре указан сокращенный интернет-адрес, то во втором параметре нужно задать базовый абсолютный адрес, на основе которого будет создан полный адрес, в виде строки или объекта URL:

```
let url = new URL('/index.php', 'http://localhost/test.php');
console.log(url.toString());
// http://localhost/index.php
```

```
let base = new URL('http://localhost/test.php');
let url = new URL('/index.php', base);
console.log(url.toString());
// http://localhost/index.php
```

Если создать объект не удалось, генерируется исключение `TypeError`. Например, указать только сокращенный интернет-адрес нельзя, нужно обязательно задать еще и базовый:

```
let url = new URL('/index.php');
// TypeError: /index.php is not a valid URL.
```

Класс URL содержит следующие свойства:

- ❑ `href` — полный интернет-адрес в виде строки:

```
let url =
 new URL('http://localhost:8080/index.php?x=5&y=3#metka');
console.log(url.href);
// http://localhost:8080/index.php?x=5&y=3#metka
```

Также можно воспользоваться методом `toString()`:

```
console.log(url.toString());
// http://localhost:8080/index.php?x=5&y=3#metka
```

- ❑ `origin` — обозначение протокола, адрес хоста и номер TCP-порта (доступно только для чтения):

```
console.log(url.origin); // http://localhost:8080
```

- ❑ `protocol` — обозначение протокола, включая символ двоеточия:

```
console.log(url.protocol); // http:
```

- ❑ `host` — адрес хоста и номер TCP-порта (если он присутствует в адресе):

```
console.log(url.host); // localhost:8080
```

- ❑ `hostname` — адрес хоста:

```
console.log(url.hostname); // localhost
```

- ❑ `port` — номер TCP-порта в виде строки или пустая строка, если порт в адресе не указан:

```
console.log(url.port); // 8080
```

- ❑ `pathname` — путь к файлу с начальным символом `/` или пустая строка, если путь не указан:

```
console.log(url.pathname); // /index.php
```

- ❑ `search` — строка GET-параметров вместе с начальным символом `?` или пустая строка, если GET-параметров в адресе нет:

```
console.log(url.search); // ?x=5&y=3
```

- ❑ `searchParams` — объект класса `URLSearchParams`, представляющий набор GET-параметров, которые присутствуют в адресе (будет описан далее):

```
console.log(url.searchParams.toString());
// x=5&y=3
```

- ❑ `hash` — имя якоря вместе с начальным символом `#` или пустая строка, если якоря в адресе нет:

```
console.log(url.hash); // #metka
```

- ❑ `username` — имя пользователя или пустая строка, если имя в адресе не указано;

- ❑ `password` — пароль пользователя или пустая строка, если пароль в адресе не указан.

Пример:

```
let ftp =
 new URL('ftp://user:123@localhost:21/index.php');
console.log(ftp.username); // user
console.log(ftp.password); // 123
```

Почти все свойства (кроме `origin`) доступны как для чтения, так и для записи:

```
let url = new URL('http://localhost');
console.log(url.toString()); // http://localhost/
url.protocol = 'https:';
url.hostname = 'test.ru';
url.port = '8080';
url.pathname = '/index.php';
url.search = '?x=5&y=3';
url.hash = '#metka';
console.log(url.toString());
// https://test.ru:8080/index.php?x=5&y=3#metka
url.host = 'localhost:8080';
console.log(url.toString());
// https://localhost:8080/index.php?x=5&y=3#metka
```

Объект `URLSearchParams`, представляющий набор GET-параметров, содержит следующие методы:

- ❑ `toString()` — возвращает все GET-параметры в виде закодированной строки:

```
let url = new URL('http://localhost/index.php?x=5&y=3');
let params = new URLSearchParams(url.search);
```

```

console.log(params.toString()); // x=5&y=3
let params2 = new URLSearchParams('?x=5&y=3');
console.log(params2.toString()); // x=5&y=3
let params3 = new URLSearchParams('x=5&y=3');
console.log(params3.toString()); // x=5&y=3
let params4 = new URLSearchParams();
console.log(params4.toString()); //

```

- `get(<Имя>)` — возвращает значение GET-параметра с указанным именем. Если GET-параметр не найден, возвращается значение `null`. Пример:

```

let params = new URLSearchParams('x=5&y=3');
console.log(params.get('x')); // 5
console.log(params.get('y')); // 3
console.log(params.get('z')); // null

```

Если GET-параметров с указанным именем несколько, метод вернет значение первого параметра:

```

let params = new URLSearchParams('x=5&x=3');
console.log(params.get('x')); // 5

```

- `getAll(<Имя>)` — возвращает массив со всеми значениями GET-параметра с указанным именем. Если параметр не найден, возвращается пустой массив. Пример:

```

let params = new URLSearchParams('x=5&x=3');
console.log(params.getAll('x')); // Array ["5", "3"]
console.log(params.getAll('y')); // Array []

```

- `keys()` — возвращает итератор, позволяющий перебрать имена всех имеющихся GET-параметров:

```

let params = new URLSearchParams('x=5&y=3');
for (let key of params.keys()) {
 console.log(key + ' => ' + params.get(key));
}
// x => 5
// y => 3

```

- `values()` — возвращает итератор, позволяющий перебрать значения всех имеющихся GET-параметров:

```

let params = new URLSearchParams('x=5&y=3');
for (let value of params.values()) {
 console.log(value);
}
// 5
// 3

```

- `entries()` — возвращает итератор, позволяющий перебрать имена и значения всех имеющихся GET-параметров:

```
let params = new URLSearchParams('x=5&y=3');
for (let pair of params.entries()) {
 console.log(pair[0] + ' => ' + pair[1]);
}
// x => 5
// y => 3
```

При использовании цикла `for...of` мы можем опустить вызов метода `entries()`:

```
let params = new URLSearchParams('x=5&y=3');
for (let pair of params) {
 console.log(pair);
 console.log(pair[0] + ' => ' + pair[1]);
}
// Array ["x", "5"]
// x => 5
// Array ["y", "3"]
// y => 3
for (let [key, value] of params) {
 console.log(key + ' => ' + value);
}
```

- ❑ `forEach(<Функция>)` — вызывает заданную функцию для каждого GET-параметра. Через первый параметр внутри этой функции доступно значение очередного GET-параметра, а через второй — его имя:

```
let params = new URLSearchParams('x=5&y=3');
params.forEach(function(value, key) {
 console.log(key + ' => ' + value);
});
// x => 5
// y => 3
```

- ❑ `has(<Имя>)` — возвращает значение `true`, если GET-параметр с указанным именем существует, и `false` — в противном случае:

```
let params = new URLSearchParams('x=5&y=3');
console.log(params.has('x')); // true
console.log(params.has('n')); // false
```

- ❑ `set(<Имя>, <Значение>)` — задает новое значение для GET-параметра с указанным именем. Если GET-параметр не существует, он будет создан. Все специальные символы кодируются. Пример:

```
let params = new URLSearchParams('x=5&y=3');
params.set('x', '12');
params.set('txt', 'строка');
console.log(params.toString());
// x=12&y=3&txt=%D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B0
console.log(params.get('txt'));
// строка
```

- ❑ `append(<Имя>, <Значение>)` — добавляет GET-параметр с указанными именем и значением, не изменяя значение одноименного существующего параметра. Все специальные символы кодируются. Пример:

```
let params = new URLSearchParams('x=5&y=3');
params.append('x', '12');
params.append('txt', 'строка');
console.log(params.toString());
// x=5&y=3&x=12&txt=%D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B0
console.log(params.get('txt'));
// строка
console.log(params.getAll('x'));
// Array ["5", "12"]
```

- ❑ `delete(<Имя>)` — удаляет все GET-параметры с указанным именем:

```
let params = new URLSearchParams('x=5&y=3&y=10');
params.delete('y');
console.log(params.toString()); // x=5
```

В листинге 25.12 мы создавали интернет-адрес с GET-параметрами следующим образом:

```
let txt1 = document.getElementById('txt1').value;
let txt2 = document.getElementById('txt2').value;
...
let url = '/ajax.php?txt1=' + encodeURIComponent(txt1);
url += '&txt2=' + encodeURIComponent(txt2);
```

Вместо передачи строки с интернет-адресом в первом параметре функции `fetch()` мы можем указать объект `URL`:

```
let txt1 = document.getElementById('txt1').value;
let txt2 = document.getElementById('txt2').value;
...
let url = new URL('/ajax.php', window.location.href);
url.searchParams.set('txt1', txt1);
url.searchParams.set('txt2', txt2);
...
let response = await fetch(url);
```

Использовать функцию `encodeURIComponent()` при этом не нужно. Все специальные символы будут закодированы. Например, если в первое поле ввести число 12, а во второе — слово "тест", то интернет-адрес будет выглядеть так (при базовом адресе **`http://localhost/test.html`**):

```
console.log(url.toString());
// http://localhost/ajax.php?txt1=12&txt2=%D1%82%D0%B5%D1%81%D1%82
```

Объект `URL` мы также можем передать методу `open()` объекта `XMLHttpRequest`, например, как в листинге 25.8:

```
var url = new URL('/ajax.php', window.location.href);
url.searchParams.set('txt1', txt1);
url.searchParams.set('txt2', txt2);
. . .
var req = new XMLHttpRequest();
req.open('GET', url, true);
```

Следует помнить, что объект `URL` доступен только в современных веб-браузерах.

#### НА ЗАМЕТКУ

Результаты при кодировании специальных символов с использованием объекта `URL` и функции `encodeURIComponent()` могут различаться. Они реализуют разные стандарты: RFC3986 (объект `URL`; <https://www.ietf.org/rfc/rfc3986.txt>) и RFC2396 (функция `encodeURIComponent()`; <https://www.ietf.org/rfc/rfc2396.txt>).

### 25.2.3. Объект запроса *Request*

В первом параметре функции `fetch()` также можно передать объект класса `Request`, представляющий готовый клиентский запрос. Формат конструктора этого класса:

```
<Объект> = new Request(<Интернет-адрес или объект Request>[, <Настройки>])
```

В первом параметре указывается интернет-адрес (в виде строки или объекта `URL`) или другой объект клиентского запроса `Request` (при этом создается его копия). Изменим код листинга 25.12, передав функции `fetch()` объект `Request`:

```
let url = new URL('/ajax.php', window.location.href);
url.searchParams.set('txt1', txt1);
url.searchParams.set('txt2', txt2);
let myRequest = new Request(url);
. . .
let response = await fetch(myRequest);
```

Во втором параметре можно передать объект со следующими основными свойствами (полный список приведен в документации):

- ❑ `method` — метод отправки данных (например, `'GET'` или `'POST'`). По умолчанию используется метод `GET`. Пример:

```
let myRequest = new Request('http://localhost/ajax.php');
console.log(myRequest.method); // GET
let myRequest2 = new Request('http://localhost/ajax.php', {
 method: 'POST'
});
console.log(myRequest2.method); // POST
```

- ❑ `headers` — заголовки запроса в виде объекта класса `Headers` или простого объекта:

```
let myRequest = new Request('http://localhost/ajax.php', {
 method: 'POST',
 headers: {'Content-Type':
 'application/x-www-form-urlencoded'}
});
```



```
for (let [key, value] of myRequest.headers) {
 console.log(key + ' => ' + value);
} // content-type => application/x-www-form-urlencoded
```

- **body** — данные для отправки методом POST в виде строки, объекта `FormData`, `URLSearchParams` или `Blob`:

```
let myRequest = new Request('http://localhost/ajax.php', {
 method: 'POST',
 headers: {'Content-Type':
 'application/x-www-form-urlencoded'},
 body: 'txt1=1&txt2=2'
});
myRequest.text().then(function(v) {
 console.log(v); // txt1=1&txt2=2
});
```

- **mode** — обозначение режима запроса: `'cors'` (кросс-доменные запросы разрешены; значение по умолчанию), `'no-cors'` (разрешены только простые кросс-доменные запросы и обычные запросы; при выполнении кросс-доменного запроса возвращается пустой объект ответа: свойство `status` будет иметь значение 0, свойство `statusText` — пустую строку, свойство `headers` — пустой объект `Headers`, свойство `body` — `null`, свойство `type` — `'opaque'`), `'same-origin'` (кросс-доменные запросы запрещены) или `'navigate'` (имитируется переход на другую страницу в процессе навигации; применяется при загрузке HTML-фрагментов);
- **redirect** — обозначение режима обработки перенаправлений: `'follow'` (перенаправления обрабатываются; значение по умолчанию), `'error'` (при перенаправлении выдается ошибка) или `'manual'` (при перенаправлении возвращается пустой объект ответа: свойство `status` будет иметь значение 0, свойство `statusText` — пустую строку, свойство `headers` — пустой объект `Headers`, свойство `body` — `null`, свойство `type` — `'opaqueredirect'`);
- **credentials** — обозначение режима отправки cookie: `'omit'` (никогда не отправлять и не получать), `'same-origin'` (отправлять только тому же сайту) или `'include'` (всегда отправлять). Значение по умолчанию зависит от веб-браузера и его версии;
- **cache** — обозначение режима кэширования полученных данных:
  - `'default'` — кэшировать и впоследствии загружать из кэша, если кэшированная копия еще актуальна (значение по умолчанию);
  - `'no-store'` — вообще не кэшировать;
  - `'reload'` — кэшировать, но впоследствии всегда загружать с сервера, даже если существует актуальная кэшированная копия;
  - `'no-cache'` — кэшировать и впоследствии перед извлечением из кэша сохраненной копии спрашивать у сервера, актуальна ли она;

- 'force-cache' — кэшировать и впоследствии всегда загружать из кэша, если там есть сохраненная копия;
- 'only-if-cached' — кэшировать и всегда загружать из кэша. Если в кэше нет сохраненной копии файла, выдавать ошибку с кодом 504 (превышено допустимое время выполнения запроса);
- `referrer` — значение HTTP-заголовка `Referer` отправляемого клиентского запроса;
- `referrerPolicy` — обозначение политики указания HTTP-заголовка `Referrer` в отправляемом клиентском запросе:
  - 'no-referrer' — заголовок `Referer` никогда не отправляется;
  - 'no-referrer-when-downgrade' — при выполнении запросов от защищенного сайта к незащищенному заголовок `Referer` не отправляется, в остальных случаях отправляется полный интернет-адрес (значение по умолчанию);
  - 'strict-origin' — при выполнении запросов от защищенного сайта к незащищенному заголовок `Referer` не отправляется, в остальных случаях отправляется адрес домена;
  - 'strict-origin-when-cross-origin' — при выполнении запросов от защищенного сайта к незащищенному заголовок `Referer` не отправляется, при запросах к другим сайтам отправляется адрес домена, при запросах к тому же сайту — полный интернет-адрес;
  - 'same-origin' — при выполнении запросов к другим сайтам заголовок `Referer` не отправляется, при запросах к тому же сайту отправляется полный интернет-адрес;
  - 'origin' — всегда отправляется адрес домена;
  - 'origin-when-cross-origin' — при выполнении запросов к другим сайтам отправляется адрес домена, при запросах к тому же сайту — полный интернет-адрес;
  - 'unsafe-url' — всегда отправляется полный интернет-адрес.

#### Пример:

```
let myRequest = new Request('http://www.somesite.ru/ajax.php', {
 method: 'POST',
 mode: 'cors',
 redirect: 'error',
 credentials: 'include',
 cache: 'no-store',
 referrer: '';
 referrerPolicy: 'no-referrer'
});
```

- `integrity` — хеш-код для проверки целостности данных, например, SHA-256.

Объект с такими же свойствами (а также с некоторыми дополнительными) можно передать во втором параметре функции `fetch()`:

```
let response = await fetch(url, {
 method: 'GET',
 mode: 'cors',
 referrerPolicy: 'origin'
});
```

С помощью одноименных свойств можно получить значения перечисленных ранее параметров после создания объекта запроса. Свойства доступны только для чтения. Получить URL-адрес позволяет свойство `url`, например:

```
let myRequest = new Request('http://localhost/ajax.php');
console.log(myRequest.url); // http://localhost/ajax.php
```

Свойство `bodyUsed` содержит значение `true`, если содержимое свойства `body` уже было прочитано, и `false` — в противном случае.

### **ОБРАТИТЕ ВНИМАНИЕ**

Значение свойства `body` можно прочитать только один раз. Попытка повторного прочтения приведет к исключению `TypeError`.

Получим содержимое свойства `body` в виде текста и выведем значение свойства `bodyUsed` до и после прочтения:

```
let myRequest = new Request('http://localhost/ajax.php', {
 method: 'POST',
 headers: {'Content-Type': 'application/x-www-form-urlencoded'},
 body: 'txt1=1&txt2=2'
});
console.log(myRequest.bodyUsed); // false
myRequest.text().then(function(v) {
 console.log(v); // txt1=1&txt2=2
 console.log(myRequest.bodyUsed); // true
});
```

Объект `Request` содержит следующие методы:

❑ `clone()` — создает и возвращает копию текущего запроса:

```
let myRequest = new Request('http://localhost/ajax.php');
let myRequest2 = myRequest.clone();
console.log(myRequest2.url); // http://localhost/ajax.php
```

Для создания копии также можно воспользоваться конструктором класса `Request`:

```
let myRequest = new Request('http://localhost/ajax.php');
let myRequest2 = new Request(myRequest);
console.log(myRequest2.url); // http://localhost/ajax.php
```

❑ `text()` — возвращает объект `Promise`, содержащий значение свойства `body` в виде простого текста:

```
let myRequest = new Request('http://localhost/ajax.php', {
 method: 'POST',
 body: 'txt1=1&txt2=2'
});
myRequest.text().then(function(v) {
 console.log(v); // txt1=1&txt2=2
});
```

- ❑ `formData()` — возвращает объект `Promise`, содержащий значение свойства `body` в виде объекта `FormData`. При ошибке генерируется исключение `TypeError`. Пример:

```
let formData = new FormData();
formData.append('txt1', '1');
formData.append('txt2', '2');
let myRequest = new Request('http://localhost/ajax.php', {
 method: 'POST',
 body: formData
});
myRequest.formData().then(function(v) {
 for (let [key, value] of v) {
 console.log(key + ' => ' + value);
 }
});
// txt1 => 1
// txt2 => 2
```

- ❑ `json()` — возвращает объект `Promise`, содержащий значение свойства `body` в виде объекта `JSON`. При ошибке генерируется исключение `SyntaxError`. Пример:

```
let myRequest = new Request('http://localhost/ajax.php', {
 method: 'POST',
 body: '{"txt1": "1", "txt2": "2"}'
});
myRequest.json().then(function(json) {
 console.log(json.txt1); // 1
 console.log(json.txt2); // 2
});
```

- ❑ `blob()` — возвращает объект `Promise`, содержащий значение свойства `body` в виде объекта `Blob`;
- ❑ `arrayBuffer()` — возвращает объект `Promise`, содержащий значение свойства `body` в виде объекта `ArrayBuffer`.

## 25.2.4. Отправка данных при выгрузке веб-страницы

Если нужно отправить на сервер небольшие (не более 64 Кб) данные, когда пользователь покидает страницу, то в объект с настройками, передаваемый вторым параметром функции `fetch()` и конструктору класса `Request`, следует добавить свойство

`keepalive` со значением `true`. Запрос должен быть отправлен внутри обработчика события `onunload` методом `POST`. Полученный от сервера ответ игнорируется.

Пример:

```
window.onunload = function() {
 fetch('/data.php', {
 method: 'POST',
 body: 'myData',
 keepalive: true
 });
};
```

В коде серверной программы `data.php` мы можем получить данные и, например, записать их в файл:

```
$data = file_get_contents('php://input');
file_put_contents('data.txt', $data, LOCK_EX | FILE_APPEND);
```

## 25.2.5. Прерывание запроса

Чтобы иметь возможность прервать выполнение запроса, нужно создать объект класса `AbortController`, затем присвоить значение свойства `signal` этого объекта одноименному свойству, принадлежащему объекту с настройками, который передается функции `fetch()` и конструктору класса `Request`. Для прерывания запроса нужно вызвать метод `abort()` объекта `AbortController`. При этом будет сгенерировано исключение `AbortError`. Пример:

```
let controller = new AbortController();
setTimeout(function() {
 controller.abort();
}, 500);
let response = await fetch(url, {
 signal: controller.signal
});
```

## 25.2.6. Класс *FormData*

В свойстве `body` объекта с настройками, передаваемого функции `fetch()` и конструктору класса `Request`, можно указать отправляемые серверной программе данные также в виде объекта класса `FormData`. При этом в запрос автоматически добавляется заголовок `Content-Type` со значением `multipart/form-data`, который также позволяет отправить серверной программе файлы. Формат конструктора класса `FormData`:

```
<Объект> = new FormData([<Ссылка на форму>])
```

Если в качестве параметра указать ссылку на форму, то в создаваемый объект будут перенесены значения из всех элементов этой формы, включая поля для выбора файлов. Эти значения будут записаны в объект под названиями, взятыми из пара-

метров `name` соответствующих элементов формы. Пример получения всех данных из формы с идентификатором `frm1`:

```
let formData = new FormData(document.getElementById('frm1'));
```

Если параметр не указан, будет создан пустой объект. Отправляемые серверной программе значения придется добавить в него вручную, задав для них соответствующие названия. Пример:

```
let formData = new FormData();
formData.append('txt1', '1');
formData.append('txt2', '2');
console.log(formData.get('txt1')); // 1
console.log(formData.get('txt2')); // 2
```

Класс `FormData` содержит следующие методы:

- `set(<Название>, <Значение>[, <Имя файла>])` — добавляет в текущий объект новое значение под указанным названием. Если значение с таким названием уже существует, оно будет заменено указанным. Пример:

```
let formData = new FormData();
formData.set('x', '5');
formData.set('x', '12');
console.log(formData.getAll('x')); // Array ["12"]
```

Во втором параметре можно указать файл в виде объекта `Blob` или `File`. В третьем параметре задается имя добавляемого файла, которое будет отправлено серверной программе;

- `append(<Название>, <Значение>[, <Имя файла>])` — добавляет в текущий объект новое значение с указанным названием, не изменяя уже существующее одноименное значение:

```
let formData = new FormData();
formData.append('x', '5');
formData.append('x', '12');
console.log(formData.getAll('x')); // Array ["5", "12"]
```

Во втором параметре можно указать файл в виде объекта `Blob` или `File`. В третьем параметре задается имя добавляемого файла, которое будет отправлено серверной программе;

- `get(<Название>)` — возвращает хранящееся в текущем объекте значение с указанным названием. Если значений с заданным названием несколько, возвращает самое первое. Если значение отсутствует, возвращается `null`. Пример:

```
let formData = new FormData();
formData.append('x', '1');
formData.append('x', '2');
console.log(formData.get('x')); // 1
console.log(formData.get('y')); // null
```

- ❑ `getAll(<Название>)` — возвращает массив со всеми значениями с указанным названием. Если значения с таким названием отсутствуют, возвращается пустой массив. Пример:

```
let formData = new FormData();
formData.append('x', '1');
formData.append('x', '2');
console.log(formData.getAll('x')); // Array ["1", "2"]
console.log(formData.getAll('y')); // Array []
```

- ❑ `keys()` — возвращает итератор, позволяющий перебрать все названия существующих значений:

```
let formData = new FormData();
formData.append('x', '5');
formData.append('y', '3');
for (let key of formData.keys()) {
 console.log(key + ' => ' + formData.get(key));
}
// x => 5
// y => 3
```

- ❑ `values()` — возвращает итератор, позволяющий перебрать все существующие значения:

```
let formData = new FormData();
formData.append('x', '5');
formData.append('y', '3');
for (let value of formData.values()) {
 console.log(value);
}
// 5
// 3
```

- ❑ `entries()` — возвращает итератор, позволяющий перебрать все существующие значения вместе с их названиями:

```
let formData = new FormData();
formData.append('x', '5');
formData.append('y', '3');
for (let pair of formData.entries()) {
 console.log(pair[0] + ' => ' + pair[1]);
}
// x => 5
// y => 3
```

При использовании цикла `for...of` мы можем опустить вызов метода `entries()`:

```
let formData = new FormData();
formData.append('x', '5');
formData.append('y', '3');
```

```
for (let pair of formData) {
 console.log(pair);
 console.log(pair[0] + ' => ' + pair[1]);
}
// Array ["x", "5"]
// x => 5
// Array ["y", "3"]
// y => 3
for (let [key, value] of formData) {
 console.log(key + ' => ' + value);
}
```

- ❑ `forEach(<Функция>)` — вызывает заданную функцию для каждого существующего в текущем объекте значения. Через первый параметр внутри функции доступно очередное значение, а через второй — его название. Пример:

```
let formData = new FormData();
formData.append('x', '5');
formData.append('y', '3');
formData.forEach(function(value, key) {
 console.log(key + ' => ' + value);
});
// x => 5
// y => 3
```

- ❑ `has(<Название>)` — возвращает `true`, если значение с указанным названием существует в текущем объекте, и `false` — в противном случае:

```
let formData = new FormData();
formData.append('x', '5');
console.log(formData.has('x')); // true
console.log(formData.has('y')); // false
```

- ❑ `delete(<Название>)` — удаляет все значения с указанным названием:

```
let formData = new FormData();
formData.append('x', '5');
formData.append('x', '12');
formData.delete('x');
console.log(formData.getAll('x')); // Array []
```

## 25.2.7. Отправка файлов

Рассмотрим пример использования класса `FormData` для отправки файлов серверной программе. Создадим страницу `index.html` (листинг 25.13), которая будет отправлять введенное значение и выбранный файл серверной программе `ajax.php` (листинг 25.14). В составе серверного ответа она получит подтверждение в формате JSON.



**Листинг 25.13. Класс FormData**

```

<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <meta name="viewport"
 content="width=device-width, initial-scale=1, shrink-to-fit=no">
 <link rel="stylesheet" type="text/css" href="css/bootstrap.min.css">
 <title>Класс FormData</title>
</head>
<body>
<div class="container my-3">
 <form action="ajax.php" method="POST" id="frm1"
 enctype="multipart/form-data" onsubmit="return false;">
 <div class="form-group">
 <input type="text" class="form-control"
 id="txt1" name="txt1" required>
 </div>
 <div class="form-group">
 <input type="file" id="file1" name="file1" required>
 </div>
 <button type="button" class="btn btn-primary"
 id="btnSend">Отправить методом POST</button>
 </form>
</div>
<div class="container my-3">
 <div id="div_ajax"></div>
</div>

<script>
async function sendReqPOST() {
 if (!window.fetch) {
 window.alert('Ваш браузер не поддерживает Fetch API');
 return;
 }
 let frm1 = document.getElementById('frm1');
 if (!frm1.checkValidity()) {
 window.alert('Поле не заполнено');
 return;
 }
 let formData = new FormData(frm1);
 document.getElementById('div_ajax').innerHTML = 'Загрузка...';
 try {
 let response = await fetch('/ajax.php', {
 method: 'POST',
 body: formData
 });
 }
}

```

```

 if (!response.ok) {
 throw new Error('Статус: ' + response.status);
 }
 let json = await response.json();
 const msg = json.txt1 + '
' + json.file1;
 document.getElementById('div_ajax').innerHTML = msg;
 frm1.reset();
 } catch(e) {
 document.getElementById('div_ajax').innerHTML = 'Ошибка: ' + e;
 }
}
document.getElementById('btnSend').onclick = sendReqPOST;
</script>
</body>
</html>

```

#### Листинг 25.14. Серверная программа ajax.php

```

<?php
// Запрещаем кэширование
header('Expires: Thu, 01 Jul 2020 01:00:00 GMT');
header('Cache-Control: no-store, no-cache, must-revalidate');
header('Pragma: no-cache');
// Указываем MIME-тип и кодировку
header('Content-Type: application/json; charset=utf-8');
if (isset($_POST['txt1'])) {
 if (isset($_FILES['file1'])) {
 if ($FILES['file1']['error'] === 0 &&
 $FILES['file1']['size'] > 0) {
 $file1 = $FILES['file1']['tmp_name'] . ' размер: ' .
 $FILES['file1']['size'] . ' название ' .
 $FILES['file1']['name'];
 // Перемещаем файл в место постоянного хранения
 // с помощью функции move_uploaded_file()
 }
 else $file1 = 'Не удалось загрузить файл';
 }
 else $file1 = 'Файл не отправлен';
 $txt1 = htmlspecialchars($_POST['txt1'], ENT_COMPAT, 'UTF-8');
 $file1 = htmlspecialchars($file1, ENT_COMPAT, 'UTF-8');
 $arr = array('txt1' => $txt1, 'file1' => $file1);
 echo json_encode($arr);
}
else {
 $arr = array('txt1' => 'Данные не получены', 'file1' => '');
 echo json_encode($arr);
}

```

Мы сделали все поля обязательными для заполнения, добавив в их теги параметр `required`. Для проверки заполнения поля и выбора файла используется метод `checkValidity()` объекта формы. Если все элементы формы успешно прошли проверку, метод вернет значение `true`, в противном случае — значение `false`:

```
let frm1 = document.getElementById('frm1');
if (!frm1.checkValidity()) {
 window.alert('Поле не заполнено');
 return;
}
```

После проверки получаем все данные из формы, передав ссылку на нее конструктору класса `FormData`:

```
let formData = new FormData(frm1);
```

Далее в объект, передаваемый функции `fetch()` вторым параметром, добавляем свойство `body` и присваиваем ему только что созданный объект `FormData`. Дополнительно указываем метод `POST` в свойстве `method` того же объекта:

```
let response = await fetch('/ajax.php', {
 method: 'POST',
 body: formData
});
```

При этом автоматически добавляется заголовок `Content-Type` со значением `multipart/form-data`, который позволяет отправить файл на сервер.

Далее проверяем успешность выполнения операции, получаем данные в формате `JSON`, выводим результат и очищаем все элементы формы, вызвав метод `reset()`:

```
frm1.reset();
```

В коде серверной программы `ajax.php` запрещаем кэширование, указываем MIME-тип и кодировку данных. Если форма отправлена, будет существовать переменная `$_POST['txt1']`. Отправленный файл доступен через переменную `$_FILES['file1']`, где `file1` — значение параметра `name` поля для выбора файла. Если файл загружен без ошибок, то переменная `$_FILES['file1']['error']` содержит значение `0`, а размер файла (`$_FILES['file1']['size']`) больше нуля.

Следует учитывать, что полученный файл сохраняется в каталоге для временных файлов, и путь к нему доступен через переменную `$_FILES['file1']['tmp_name']`. При поступлении следующего клиентского запроса серверная программа автоматически очистит этот каталог. Поэтому сразу же следует сохранить принятый файл, переместив его на место постоянного хранения с помощью функции `move_uploaded_file()`. В нашем примере мы просто получаем информацию о загруженном файле, а затем отправляем ее в составе ответа в формате `JSON`.

## 25.2.8. Объект ответа *Response*

Функция `fetch()` возвращает объект `Promise`, содержащий в себе объект класса `Response`, который представляет полученный серверный ответ. В случае необходи-

мости можно создать объект `Response` вручную с помощью конструктора следующего формата:

```
<Объект> = new Response([<Тело ответа>[, <Настройки>]])
```

Все параметры являются необязательными. В первом параметре можно задать тело ответа в виде строки, объекта `FormData`, `URLSearchParams`, `Blob`, `BufferSource` или `ReadableStream`. Во втором параметре указывается объект со следующими свойствами:

- ☐ `status` — код статуса ответа (например, 200);
- ☐ `statusText` — статус ответа в виде строки (например, 'ОК');
- ☐ `headers` — заголовки ответа сервера в виде простого объекта или объекта класса `Headers`.

Пример:

```
let params = new URLSearchParams('x=5&y=3');
let myResponse = new Response(params, {
 'status': 404,
 'statusText': 'Not Found'
});
console.log(myResponse.status); // 404
console.log(myResponse.statusText); // Not Found
myResponse.text().then(function(v) {
 console.log(v); // x=5&y=3
});
```

Объект `Response` содержит следующие свойства (доступны только для чтения):

- ☐ `status` — код статуса ответа (например, 200);
- ☐ `statusText` — статус ответа в виде строки (например, 'ОК');
- ☐ `ok` — значение `true`, если код статуса ответа в пределах от 200 до 299, и `false` — в противном случае;
- ☐ `headers` — заголовки ответа в виде объекта `Headers`;
- ☐ `redirected` — значение `true`, если выполнялись перенаправления, и `false` — в противном случае;
- ☐ `url` — интернет-адрес серверной программы, приславшей ответ;
- ☐ `type` — обозначение типа ответа: `'basic'` (обычный), `'cors'` (при выполнении кросс-доменного запроса), `'error'` (при ошибке), `'opaque'` (при выполнении кросс-доменного запроса, если настройке `mode` было дано значение `'no-cors'`) или `'opaqueredirect'` (при перенаправлении, если настройке `redirect` было дано значение `'manual'`);
- ☐ `bodyUsed` — значение `true`, если значение свойства `body` уже было прочитано, и `false` — в противном случае.

**ОБРАТИТЕ ВНИМАНИЕ**

Значение свойства `body` ответа можно прочитать только один раз. Попытка повторного прочтения приведет к исключению `TypeError`. Если хотите прочитать ответ в разных форматах, предварительно создайте копию объекта ответа.

Объект `Response` содержит следующие методы:

- ❑ `clone()` — создает и возвращает копию текущего ответа;
- ❑ `text()` — возвращает объект `Promise`, содержащий значение свойства `body` в виде простого текста:

```
let params = new URLSearchParams('x=5&y=3');
let myResponse = new Response(params);
myResponse.text().then(function(v) {
 console.log(v); // x=5&y=3
});
```

- ❑ `formData()` — возвращает объект `Promise`, содержащий значение свойства `body` в виде объекта `FormData`. При ошибке генерируется исключение `TypeError`. Пример:

```
let formData = new FormData();
formData.append('txt1', '1');
formData.append('txt2', '2');
let myResponse = new Response(formData);
myResponse.formData().then(function(v) {
 for (let [key, value] of v) {
 console.log(key + ' => ' + value);
 }
});
// txt1 => 1
// txt2 => 2
```

- ❑ `json()` — возвращает объект `Promise`, содержащий значение свойства `body` в виде объекта `JSON`. При ошибке генерируется исключение `SyntaxError`. Пример:

```
let myResponse = new Response('{ "txt1": "1", "txt2": "2" }');
myResponse.json().then(function(json) {
 console.log(json.txt1); // 1
 console.log(json.txt2); // 2
});
```

- ❑ `blob()` — возвращает объект `Promise`, содержащий значение свойства `body` в виде объекта `Blob`.

Загрузим изображение с сервера, вернем его в виде объекта `Blob`, получим его адрес в памяти с помощью статического метода `createObjectURL()` из класса `URL` и присвоим этот адрес параметру `src` тега `<img>` с идентификатором `img1`, чтобы вывести загруженное изображение на странице:

```

. . .
```

```
fetch('/photo2.jpg')
 .then(function(response) {
 return response.blob();
 })
 .then(function(blob) {
 let url = URL.createObjectURL(blob);
 document.getElementById('img1').src = url;
 });
```

- ❑ `arrayBuffer()` — возвращает объект `Promise`, содержащий значение свойства `body` в виде объекта `ArrayBuffer`.

## 25.2.9. Класс *Headers*

Класс `Headers` описывает HTTP-заголовки клиентского запроса или серверного ответа. Получить доступ к заголовкам позволяет свойство `headers` объектов `Request` и `Response`. Формат конструктора класса `Headers`:

<Объект> = new Headers([<Объект с заголовками>])

В параметре можно указать простой объект с заголовками и их значениями или другой объект `Headers` (в последнем случае создается копия указанного объекта):

```
let headers = new Headers({
 'Content-Type': 'application/x-www-form-urlencoded'
});
console.log(headers.get('Content-Type'));
// application/x-www-form-urlencoded
let headers2 = new Headers(headers);
console.log(headers2.get('Content-Type'));
// application/x-www-form-urlencoded
```

Класс `Headers` содержит следующие методы:

- ❑ `set(<Имя>, <Значение>)` — добавляет в текущий объект заголовок с указанными именем и значением. Если такой заголовок существует, его значение будет заменено заданным. Пример:

```
let headers = new Headers();
headers.set('Accept-Encoding', 'deflate');
headers.set('Accept-Encoding', 'gzip');
console.log(headers.get('Accept-Encoding')); // gzip
```

- ❑ `append(<Имя>, <Значение>)` — добавляет в текущий объект заголовок с указанными именем и значением, не изменяя значение уже существующего одноименного заголовка:

```
let headers = new Headers();
headers.append('Accept-Encoding', 'deflate');
headers.append('Accept-Encoding', 'gzip');
console.log(headers.get('Accept-Encoding')); // deflate,gzip
```

- ❑ `get(<Имя>)` — возвращает значение заголовка с указанным именем (его регистр не важен). Если такой заголовок отсутствует, возвращается значение `null`. Пример:

```
let headers = new Headers({
 'Content-Type': 'application/x-www-form-urlencoded'
});
console.log(headers.get('Content-Type'));
// application/x-www-form-urlencoded
console.log(headers.get('content-type'));
// application/x-www-form-urlencoded
console.log(headers.get('Content-Length'));
// null
```

Если в текущем объекте существует несколько заголовков с указанным именем, будет возвращена строка, содержащая все их значения через запятую:

```
let headers = new Headers();
headers.append('Accept-Encoding', 'deflate');
headers.append('Accept-Encoding', 'gzip');
console.log(headers.get('Accept-Encoding'));
// deflate,gzip
```

- ❑ `keys()` — возвращает итератор, позволяющий перебрать все имена заголовков:

```
let headers = new Headers({
 'Content-Type': 'image/jpeg',
 'Content-Length': 15012
});
for (let key of headers.keys()) {
 console.log(key + ' => ' + headers.get(key));
}
// content-length => 15012
// content-type => image/jpeg
```

- ❑ `values()` — возвращает итератор, позволяющий перебрать все значения заголовков:

```
let headers = new Headers({
 'Content-Type': 'image/jpeg',
 'Content-Length': 15012
});
for (let value of headers.values()) {
 console.log(value);
}
// 15012
// image/jpeg
```

- ❑ `entries()` — возвращает итератор, позволяющий перебрать все имена и значения заголовков:

```
let headers = new Headers({
 'Content-Type': 'image/jpeg',
 'Content-Length': 15012
});
for (let pair of headers.entries()) {
 console.log(pair[0] + ' => ' + pair[1]);
}
// content-length => 15012
// content-type => image/jpeg
```

При использовании цикла `for...of` мы можем опустить вызов метода `entries()`:

```
for (let pair of headers) {
 console.log(pair);
 console.log(pair[0] + ' => ' + pair[1]);
}
// Array ["content-length", "15012"]
// content-length => 15012
// Array ["content-type", "image/jpeg"]
// content-type => image/jpeg
for (let [key, value] of headers) {
 console.log(key + ' => ' + value);
}
```

- ❑ `forEach(<Функция>)` — вызывает указанную функцию для каждого заголовка. Через первый параметр внутри функции доступно значение очередного заголовка, а через второй — его имя:

```
let headers = new Headers({
 'Content-Type': 'image/jpeg',
 'Content-Length': 15012
});
headers.forEach(function(value, key) {
 console.log(key + ' => ' + value);
});
// content-length => 15012
// content-type => image/jpeg
```

- ❑ `has(<Имя>)` — возвращает значение `true`, если заголовок с указанным именем существует, и `false` — в противном случае:

```
let headers = new Headers({
 'Content-Type': 'image/jpeg'
});
console.log(headers.has('Content-Type')); // true
console.log(headers.has('Content-Length')); // false
```

- ❑ `delete(<Имя>)` — удаляет все заголовки с указанным именем:

```
let headers = new Headers({
 'Content-Type': 'image/jpeg',
```



```
'Content-Length': 15012
});
headers.delete('Content-Length');
console.log(headers.has('Content-Type')); // true
console.log(headers.has('Content-Length')); // false
```

Выведем в консоль заголовки, отправленные сервером вместе с файлом:

```
fetch('/photo.jpg')
 .then(function(response) {
 for (let [key, value] of response.headers) {
 console.log(key + ' => ' + value);
 }
 });
```

### НА ЗАМЕТКУ

Не все заголовки можно добавить в клиентский запрос, и не все заголовки можно извлечь из серверного ответа. Некоторые заголовки обрабатываются только веб-браузером. Полный список таких заголовков приведен в документации.

## 25.2.10. Кросс-доменные запросы

Если мы со страницы <http://localhost/test.html> отправим запрос по интернет-адресу <http://site1/ajax.php>, то получим следующее сообщение об ошибке:

```
TypeError: NetworkError when attempting to fetch resource
```

Таким образом, по умолчанию кросс-доменные запросы запрещены, и мы можем отправить запрос только на тот же самый домен. Чтобы выполнить кросс-доменный запрос, нужно свойству `mode` объекта, передаваемого вторым параметром функции `fetch()` и конструктору класса `Request`, дать значение `'cors'` (разрешив тем самым кросс-доменные запросы). Вот так:

```
let url = 'http://site1/ajax.php?txt1=' + encodeURIComponent(txt1);
url += '&txt2=' + encodeURIComponent(txt2);
let response = await fetch(url, {
 mode: 'cors'
});
```

Однако этого недостаточно, ведь `'cors'` — значение по умолчанию настройки `mode`. Нужно дополнительно соблюсти политику *CORS* (Cross-Origin Resource Sharing, совместное использование ресурсов между разными источниками).

Кросс-доменные запросы делятся на два типа: *простые* и *прочие*. Простой запрос должен удовлетворять следующим основным условиям:

- ☐ метод отправки данных — GET, POST или HEAD;
- ☐ присутствуют заголовки `Accept`, `Accept-Language`, `Content-Language`, `Content-Type` (со значением `application/x-www-form-urlencoded`, `multipart/form-data` или `text/plain`) и некоторые другие, а также заголовки, добавляемые в запрос самим веб-браузером.

При выполнении простого запроса веб-браузер дополнительно добавит в отправляемый запрос заголовок `Origin` с адресом домена, например:

```
Origin: http://localhost
```

Серверная программа должна отправить в составе ответа заголовок `Access-Control-Allow-Origin`, значением которого может быть домен из заголовка запроса `Origin`:

```
header('Access-Control-Allow-Origin: http://localhost');
```

или символ `*`, означающий все домены:

```
header('Access-Control-Allow-Origin: *');
```

В нашем примере отправляется простой запрос, поэтому серверной программе, расположенной по интернет-адресу **`http://site1/ajax.php`**, достаточно добавить в ответ заголовок `Access-Control-Allow-Origin`, например, после заголовка с MIME-типом, и кросс-доменный запрос будет успешно выполнен:

```
// Указываем MIME-тип и кодировку
header('Content-Type: application/json; charset=utf-8');
// Добавляем заголовок Access-Control-Allow-Origin с адресом домена,
// с которого пришел запрос
header('Access-Control-Allow-Origin: http://localhost');
```

Если запрос не является простым, то веб-браузер выполнит предварительный запрос методом `OPTIONS`. В этом запросе будут следующие заголовки:

- ❑ `Origin` — с адресом текущего домена:

```
Origin: http://localhost
```

- ❑ `Access-Control-Request-Method` — с методом отправки данных:

```
Access-Control-Request-Method: GET
```

- ❑ `Access-Control-Request-Headers` — с именами заголовков, которые будут отправлены в составе основного запроса, через запятую:

```
Access-Control-Request-Headers: x-requested-with
```

В ответ сервер должен отправить следующие заголовки:

- ❑ `Access-Control-Allow-Origin` — с названием домена из заголовка запроса `Origin` или символом `*`:

```
$allowed_domains = array(
 'http://localhost'
);
if (!isset($_SERVER['HTTP_ORIGIN'])) {
 if (!in_array($_SERVER['HTTP_ORIGIN'], $allowed_domains))
 exit();
 header('Access-Control-Allow-Origin: ' .
 $_SERVER['HTTP_ORIGIN']);
}
```

- ❑ `Access-Control-Allow-Methods` — с разрешенными методами отправки данных (через запятую):

```
header('Access-Control-Allow-Methods: GET, OPTIONS');
```

- ❑ `Access-Control-Allow-Headers` — с именами разрешенных заголовков (через запятую):

```
header('Access-Control-Allow-Headers: ' .
 'X-Requested-With, Content-Type');
```

- ❑ `Access-Control-Max-Age` — задает время в секундах, в течение которого можно кэшировать ответ на запрос `OPTIONS` без отправки другого запроса `OPTIONS`. Нельзя задать значение больше, чем разрешено веб-браузером. Например, так можно указать время, равное 10 минутам:

```
header('Access-Control-Max-Age: 600');
```

- ❑ `Access-Control-Allow-Credentials` — если настройка `credentials` объекта запроса имеет значение `'include'`, то в этом заголовке нужно указать значение `true`. Причем в заголовке `Access-Control-Allow-Origin` обязательно должен быть указан домен из заголовка запроса `Origin`, в противном случае будет ошибка. Пример:

```
header('Access-Control-Allow-Credentials: true');
```

Если предварительный запрос выполнен успешно, то веб-браузер отправит обычный запрос, указав заголовок `Origin` с адресом домена. Сервер должен вернуть заголовок `Access-Control-Allow-Origin`. Если настройка `credentials` объекта запроса имеет значение `'include'`, то нужно дополнительно отправить заголовок `Access-Control-Allow-Credentials` со значением `true`.

Рассмотрим пример выполнения кросс-доменного запроса. Создадим страницу <http://localhost/test.html> (листинг 25.15), которая будет отправлять введенные в форму данные серверной программе <http://site1/ajax.php> (листинг 25.16) и получать от нее подтверждение в формате JSON. Чтобы запрос не был простым, укажем заголовок `X-Requested-With`, также произведем отправку и получение cookie.

#### Листинг 25.15. Веб-страница <http://localhost/test.html>

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <meta name="viewport"
 content="width=device-width, initial-scale=1, shrink-to-fit=no">
 <link rel="stylesheet" type="text/css" href="css/bootstrap.min.css">
 <title>Кросс-доменные запросы</title>
</head>
<body>
<div class="container my-3">
```

```
<form action="ajax.php" method="GET" onsubmit="return false;">
 <div class="form-group">
 <input type="text" class="form-control" id="txt1">
 </div>
 <button type="button" class="btn btn-primary"
 id="btnSend">Отправить методом GET</button>
</form>
</div>
<div class="container my-3">
 <div id="div_ajax"></div>
</div>

<script>
async function sendReqGET() {
 if (!window.fetch) {
 window.alert('Ваш браузер не поддерживает Fetch API');
 return;
 }
 let txt1 = document.getElementById('txt1').value;
 if (txt1 === '') {
 window.alert('Поле не заполнено');
 return;
 }
 let url = 'http://site1/ajax.php?txt1=' + encodeURIComponent(txt1);
 document.getElementById('div_ajax').innerHTML = 'Загрузка...';
 try {
 let response = await fetch(url, {
 mode: 'cors',
 headers: {'X-Requested-With': 'fetch'},
 credentials: 'include'
 });
 if (!response.ok) {
 throw new Error('Статус: ' + response.status);
 }
 let json = await response.json();
 const msg = json.txt1 + '
' + json.myCookie;
 document.getElementById('txt1').value = '';
 document.getElementById('div_ajax').innerHTML = msg;
 } catch(e) {
 document.getElementById('div_ajax').innerHTML = 'Ошибка: ' + e;
 }
}
document.getElementById('btnSend').onclick = sendReqGET;
</script>
</body>
</html>
```

**Листинг 25.16. Серверная программа `http://site1/ajax.php`**

```

<?php
$allowed_domains = array(
 'http://localhost'
);
if (isset($_SERVER['HTTP_ORIGIN'])) {
 if (!in_array($_SERVER['HTTP_ORIGIN'], $allowed_domains))
 exit();
 header('Access-Control-Allow-Origin: ' .
 $_SERVER['HTTP_ORIGIN']);
 header('Access-Control-Allow-Credentials: true');
}
if ($_SERVER['REQUEST_METHOD'] == 'OPTIONS') {
 header('Access-Control-Allow-Methods: GET, OPTIONS');
 header('Access-Control-Allow-Headers: ' .
 'X-Requested-With, Content-Type');
 header('Content-Type: application/json; charset=utf-8');
 header('Content-Length: 0');
 exit();
}
// Запрещаем кэширование
header('Expires: Thu, 01 Jul 2020 01:00:00 GMT');
header('Cache-Control: no-store, no-cache, must-revalidate');
header('Pragma: no-cache');
// Указываем MIME-тип и кодировку
header('Content-Type: application/json; charset=utf-8');
if (isset($_GET['txt1'])) {
 if (isset($_COOKIE['myCookie'])) {
 $myCookie = $_COOKIE['myCookie'];
 }
 else $myCookie = '';
 setcookie('myCookie', time());
 $txt1 = htmlspecialchars($_GET['txt1'], ENT_COMPAT, 'UTF-8');
 $arr = array('txt1' => $txt1, 'myCookie' => $myCookie);
 echo json_encode($arr);
}
else {
 $arr = array('txt1' => 'Данные не получены', 'myCookie' => '');
 echo json_encode($arr);
}

```

Чтобы иметь возможность выполнять кросс-доменные запросы, при создании запроса мы передаем функции `fetch()` объект настроек со свойством `mode`, имеющим значение `'cors'`. На самом деле `'cors'` является значением по умолчанию, и настройку `mode` можно вообще не указывать, но явное лучше неявного. Чтобы успеш-

но отправлять и принимать cookie, также указываем настройку `credentials` со значением `'include'`:

```
let response = await fetch(url, {
 mode: 'cors',
 headers: {'X-Requested-With': 'fetch'},
 credentials: 'include'
});
```

Так как мы добавляем заголовок `X-Requested-With`, наш запрос не является простым, поэтому сначала веб-браузер отправит предварительный запрос методом `OPTIONS`.

Если веб-браузер отправит заголовок `Origin`, то в коде серверной программы будет доступна переменная `$_SERVER['HTTP_ORIGIN']`. Мы должны вернуть заголовок `Access-Control-Allow-Origin` с тем же значением, что и в заголовке `Origin`. Символ `*` в данном случае указывать нельзя по двум причинам: во-первых, мы работаем с cookie, во-вторых, злоумышленник может украсть данные сессии. Поэтому, прежде чем отправить заголовок с разрешением, мы проверяем наличие домена в массиве `$allowed_domains`. Если домен в массиве разрешенных доменов не найден, просто завершаем выполнение программы. Чтобы успешно отправлять и принимать cookie, также указываем заголовок `Access-Control-Allow-Credentials` со значением `true`.

При отправке веб-браузером предварительного запроса переменная `$_SERVER['REQUEST_METHOD']` будет содержать значение `'OPTIONS'`. В этом случае отправляем два заголовка: `Access-Control-Allow-Methods` (с поддерживаемыми методами) и `Access-Control-Allow-Headers` (с именами поддерживаемых заголовков). После чего отправляем заголовки `Content-Type` и `Content-Length` со значением `0` и завершаем выполнение программы.

Если запрос обычный, запрещаем кэширование, указываем MIME-тип и кодировку данных. Если веб-браузер отправил cookie `myCookie`, будет существовать переменная `$_COOKIE['myCookie']`. Получаем значение этого cookie и сразу же посредством функции `setcookie()` записываем в него новое значение — число секунд, прошедшее с 1 января 1970 г., которое возвращает функция `time()`. И наконец, отправляем ранее полученное из cookie значение в составе JSON-ответа.

#### **ПРИМЕЧАНИЕ**

В современных веб-браузерах объект `XMLHttpRequest` также использует правила политики CORS при кросс-доменных запросах.



## ГЛАВА 26

# Сборка веб-сайтов

В этой главе мы рассмотрим готовые пакеты для Node.js, которые позволяют выполнить различные операции с файлами разрабатываемого веб-сайта. Например, пакет `node-sass` служит для трансляции таблиц стилей, записанных на языках SCSS и SASS, в язык CSS, "понимаемый" веб-браузером. А пакет `Webpack` может полностью автоматизировать процесс сборки сайта.

### 26.1. Файл `.browserslistrc`

Конфигурационный файл `.browserslistrc` хранит список веб-браузеров, которые должен поддерживать разрабатываемый сайт и который используется различными утилитами и пакетами (например, `autoprefixer`). Мы воспользуемся готовым файлом из состава библиотеки `Bootstrap 4.6` (листинг 26.1, этот файл также доступен по интернет-адресу <https://github.com/twbs/bootstrap/blob/v4.6.0/.browserslistrc>).

#### Листинг 26.1. Файл `.browserslistrc`

```
https://github.com/browserslist/browserslist#readme
```

```
>= 1%
last 1 major version
not dead
Chrome >= 45
Firefox >= 38
Edge >= 12
Explorer >= 10
iOS >= 9
Safari >= 9
Android >= 4.4
Opera >= 30
```

## 26.2. Сборка CSS- и SCSS-файлов

Создадим следующую структуру каталогов и файлов:

```
C:\book\p4\
 build\
 dist\
 css\
 scss\
 src\
 .browserslistrc
 package.json
```

Содержимое файла `.browserslistrc` приведено в листинге 26.1, содержимое файла `package.json` — в листинге 26.2.

### Листинг 26.2. Файл `package.json`

```
{
 "name": "p4",
 "version": "1.0.0",
 "main": "index.js",
 "scripts": {
 },
 "keywords": [],
 "author": "",
 "license": "MIT",
 "description": ""
}
```

### 26.2.1. Добавление вендорных префиксов

Пакет `autoprefixer` автоматически добавляет вендорные префиксы. Для установки этого пакета, а также необходимых ему для работы утилит `postcss` и `postcss-cli` выполняем следующую команду:

```
C:\book\p4>npm install postcss postcss-cli autoprefixer --save-dev
```

В файле `package.json` в секцию `devDependencies` будут добавлены следующие строки:

```
"devDependencies": {
 "autoprefixer": "^10.2.6",
 "postcss": "^8.3.0",
 "postcss-cli": "^8.3.1"
}
```

В секцию `scripts` файла `package.json` добавляем следующий скрипт:

```
"css-prefix": "postcss --config build/postcss.config.js",
--replace \"dist/css/*.css\" \"!dist/css/*.min.css\""
```



С помощью ключа `--config` указываем местоположение конфигурационного файла `postcss.config.js`. С помощью ключа `--replace` сообщаем, что нужно преобразовать все CSS-файлы из каталога `C:\book\p4\dist\css`, кроме файлов с расширением `min.css`, и перезаписать их.

Внутри каталога `C:\book\p4\build` создаем файл `postcss.config.js` со следующим содержанием:

```
module.exports = (ctx) => ({
 map: false,
 plugins: {
 autoprefixer: {
 cascade: false
 }
 }
});
```

Создадим в каталоге `C:\book\p4\dist\css` файл `main.css` со следующим содержанием:

```
#div1 {
 display: flex;
}
```

Запустим преобразование с помощью команды:

```
C:\book\p4>npm run css-prefix
```

В результате содержимое файла `main.css` будет выглядеть так:

```
#div1 {
 display: -ms-flexbox;
 display: flex;
}
```

## 26.2.2. Сжатие CSS-файлов

Сжать CSS-файл позволяет пакет `clean-css-cli`. Для его установки выполняем следующую команду:

```
C:\book\p4>npm install clean-css-cli --save-dev
```

В файле `package.json` в секцию `devDependencies` будет добавлена строка:

```
"clean-css-cli": "^5.3.0"
```

В секцию `scripts` файла `package.json` добавляем такой скрипт:

```
"css-minify": "cleancss -O1 --format breakWith=lf --source-map 🐞
--source-map-inline-sources --output dist/css/main.min.css 🐞
dist/css/main.css"
```

Запустим преобразование с помощью команды:

```
C:\book\p4>npm run css-minify
```

В результате в каталоге `C:\book\p4\dist\css` будет создан файл `main.min.css` со следующим содержимым:

```
#div1{display:-ms-flexbox;display:flex}
/*# sourceMappingURL=main.min.css.map */
```

Дополнительно там же будет создан файл `main.min.css.map`.

### 26.2.3. SCSS-транслятор node-sass

Язык SCSS служит для написания таблиц стилей, предоставляя расширенные по сравнению с CSS инструменты (переменные, функции, примеси и др.); веб-браузерами он не поддерживается, поэтому написанные на нем таблицы стилей следует транслировать в CSS.

Сделать это позволяет программа `node-sass`. Установим ее:

```
C:\book\p4>npm install node-sass --save-dev
```

В файле `package.json` в секцию `devDependencies` будет добавлена строка:

```
"node-sass": "^6.0.0"
```

Команда, запускающая трансляцию SCSS-кода посредством `node-sass`, имеет следующий формат:

```
node-sass [<Ключи>] <Путь к SCSS-файлу>[<Путь к CSS-файлу>]
```

Будучи вызвана без ключей, программа транслирует SCSS-файл с указанным в первом параметре путем и сохраняет результат в CSS-файле, путь к которому задан вторым параметром. Например, так записывается команда трансляции файла `C:\book\p4\scss\main.scss` в `C:\book\p4\dist\css\main.css`:

```
node-sass scss/main.scss dist/css/main.css
```

Также можно указать следующие основные командные ключи:

❑ `--output <Путь>` (или `-o <Путь>`) — задает путь до каталога, в котором будут сохраняться результирующие CSS-файлы. В случае указания этого ключа в первом параметре можно передать путь к каталогу, в котором хранятся подлежащие трансляции SCSS-файлы. Второй параметр при этом не указывается. Пример:

```
node-sass -output dist/css/ scss/
```

❑ `--output-style <Стиль>` — задает стиль итогового CSS-файла:

- `nested` — вложенный стиль (с помощью отступов показывает вложенность правил; значение по умолчанию);
- `expanded` — развернутый стиль (файл `bootstrap.css` имеет именно этот стиль);
- `compact` — компактный стиль (все определения стилей на одной строке);
- `compressed` — сжатый стиль (все правила на одной строке без пробелов; файл `bootstrap.min.css` имеет именно этот стиль).

Пример указания сжатого стиля:

```
node-sass --output-style compressed --output dist/css/ scss/
```

- ❑ `--include-path <Путь>` — задает путь поиска импортируемых таблиц стилей:

```
node-sass -o dist/css/ --include-path node_modules/ scss/main.scss
```

- ❑ `--source-map true` — если указан со значением `true`, то наряду с CSS-файлом будет создан файл с расширением `css.map` (по умолчанию файл не создается):

```
node-sass -o dist/css/ --source-map true scss/main.scss
```

Вместо значения `true` можно передать путь к каталогу, в котором будут сохраняться файлы с расширением `css.map`:

```
node-sass -o dist/css/ --source-map dist/css/ scss/main.scss
```

- ❑ `--source-map-contents true` — если указан со значением `true`, то внутри файла с расширением `css.map` будет помещено содержимое файла с исходным SCSS-кодом:

```
node-sass -o dist/css/ --source-map true ↵
--source-map-contents true scss/main.scss
```

- ❑ `--source-map-embed true` — если указан со значением `true`, то внутри результирующего CSS-файла будет помещено закодированное содержимое файла `css.map`:

```
node-sass -o dist/css/ --source-map-embed true scss/main.scss
```

- ❑ `--source-map-root <Путь>` — задает значение для свойства `sourceRoot` внутри файла с расширением `css.map`;

- ❑ `--omit-source-map-url true` (или `-x true`) — если указан со значением `true`, то внутри CSS-файла не будет помещаться комментарий формата:

```
/*# sourceMappingURL=<Имя>.css.map */
```

Пример указания флага:

```
node-sass -o dist/css/ --source-map true ↵
--omit-source-map-url true scss/main.scss
```

- ❑ `--source-comments true` — если указан со значением `true`, то внутри CSS-файла будут помещены комментарии с номером строки и именем файла с исходным SCSS-кодом, например:

```
/* line 4, scss/main.scss */
body {
 color: #ffffff;
 background-color: #000000; }
```

- ❑ `--precision <Число>` — задает максимальное число цифр после точки у вещественных чисел, которыми выражаются результаты вычислений (значение по умолчанию: 5). Укажем шесть цифр:

```
node-sass -o dist/css/ --precision 6 scss/main.scss
```

- ❑ `--indent-type <space | tab>` — указывает символ, который будет использоваться внутри CSS-файла для создания отступов. После флага можно передать значение `space` (пробел; значение по умолчанию) или `tab` (табуляция):

```
node-sass -o dist/css/ --indent-type tab scss/main.scss
```

- ❑ `--indent-width <Число>` — задает число символов для создания отступов внутри CSS-файла. После флага можно передать число до 10. Значение по умолчанию: 2. Укажем три пробела:

```
node-sass -o dist/css/ --indent-type space --indent-width 3 scss/main.scss
```

- ❑ `--linefeed <cr | crlf | lf | lfcr>` — задает комбинацию символов для выполнения переводов строк внутри CSS-файла. Можно указать значение `crlf` (`\r\n`), `cr` (`\r`), `lf` (`\n`; значение по умолчанию) или `lfcr` (`\n\r`). Укажем комбинацию `\r\n`:

```
node-sass -o dist/css/ --linefeed crlf scss/main.scss
```

- ❑ `--quiet true` (или `-q true`) — если указан со значением `true`, то информация о процессе преобразования в консоли выводиться не будет (за исключением сообщений об ошибках). По умолчанию вывод выглядит так:

```
> node-sass -o dist/css/ --quiet false scss/main.scss
```

```
Rendering Complete, saving .css file...
```

```
Wrote CSS to C:\book\p4\dist\css\main.css
```

```
C:\book\p4>
```

Если мы укажем флаг со значением `true`, то получим такой результат:

```
> node-sass -o dist/css/ --quiet true scss/main.scss
```

```
C:\book\p4>
```

- ❑ `--indented-syntax true` (или `-i true`) — если указан со значением `true`, то программа `node-sass` будет трактовать исходные файлы как написанные на языке SASS (по умолчанию они трактуются как написанные на SCSS);

- ❑ `--recursive false` (или `-r false`) — если указан со значением `false`, то рекурсивный просмотр всех вложенных каталогов и трансляция хранящихся в них SCSS-файлов выполняться не будет (по умолчанию рекурсивный просмотр выполняется). После исполнения следующей команды файлы с исходным SCSS-кодом, расположенные во вложенных каталогах, транслироваться не будут.

```
node-sass -r false -o dist/css/ scss/
```

Чтобы изменения исходных SCSS-файлов отслеживались кодом и автоматически осуществлялась их трансляция в CSS, при наборе команды следует указать ключ `--watch` (или `-w`).

Для получения полного описания команды укажите ключ `--help`:

```
node-sass --help
```

В секцию `scripts` файла `package.json` добавляем следующий скрипт:

```
"sass": "node-sass --output-style expanded --precision 6 -o dist/css/ scss/"
```

В каталоге `C:\book\p4\scss` создаем файл `main.scss` с содержимым из листинга 26.3.

#### Листинг 26.3. Содержимое файла `main.scss`

```
$black: #000000;
$white: #ffffff;

body {
 color: $white;
 background-color: $black;
}
#div1 {
 display: flex;
}
```

Выполним трансляцию подачей команды:

```
C:\book\p4>npm run sass
```

В результате в каталоге `C:\book\p4\dist\css` будет создан файл `main.css` со следующим содержимым:

```
body {
 color: #ffffff;
 background-color: #000000;
}

#div1 {
 display: flex;
}
```

Теперь можно добавить вендорные префиксы с помощью команды:

```
npm run css-prefix
```

а затем сжать получившийся CSS-файл командой:

```
npm run css-minify
```

## 26.3. Пакет *npm-run-all*: запуск нескольких скриптов

В предыдущем примере мы запускали последовательно три скрипта:

```
npm run sass
npm run css-prefix
npm run css-minify
```

Пакет `npm-run-all` позволяет выполнить сразу несколько скриптов как последовательно, так и параллельно (если это возможно). Установим пакет:

```
C:\book\p4>npm install npm-run-all --save-dev
```

В файле `package.json` в секцию `devDependencies` будет добавлена строка:

```
"npm-run-all": "^4.1.5"
```

В секцию `scripts` файла `package.json` добавляем следующий скрипт:

```
"css": "npm-run-all sass css-prefix css-minify"
```

Теперь все три операции можно запустить набором одной команды:

```
C:\book\p4>npm run css
```

## 26.4. Сборщик проектов Webpack

Пакет `Webpack` выполняет обработку JavaScript-файлов, а с помощью дополнительных программ способен обрабатывать файлы других типов. Установим `Webpack` и необходимую ему для работы утилиту `webpack-cli` командой:

```
C:\book\p4>npm install webpack webpack-cli --save-dev
```

В файле `package.json` в секцию `devDependencies` будут добавлены две строки:

```
"webpack": "^5.38.1",
"webpack-cli": "^4.7.0"
```

В секцию `scripts` файла `package.json` записываем следующие скрипты:

```
"dev-pack": "webpack --mode development",
"build-pack": "webpack --mode production"
```

Скрипт `dev-pack` создает сборку в режиме разработки без сжатия, а скрипт `build-pack` — окончательную сборку со сжатием, которую можно использовать для эксплуатации.

### 26.4.1. Создание файла конфигурации

Прежде чем запускать скрипты, нужно в корне пакета (в нашем случае — в каталоге `C:\book\p4`) создать файл конфигурации `webpack.config.js`. Содержимое файла приведено в листинге 26.4.

#### Листинг 26.4. Содержимое файла `C:\book\p4\webpack.config.js`

```
const path = require('path');

module.exports = {
 entry: {
 main: './src/index.js'
 },
}
```

```
output: {
 filename: 'index.bundle.js',
 path: path.resolve(__dirname, 'dist')
},
plugins: [],
module: {}
};
```

С помощью свойства `entry` задается точка входа в приложение. В качестве значения указывается строка, массив строк или объект, что позволяет задать несколько точек входа. Путь к файлу указывается относительно самого файла `webpack.config.js`. Свойство `output` задает имя (`filename`) и путь (`path`) к файлу, в который будет записан результат. Мы запишем результат в файл `C:\book\p4\dist\index.bundle.js`. Свойства `plugins` и `module` позволяют подключить дополнительные плагины и пакеты.

#### **НА ЗАМЕТКУ**

С помощью ключа `--config` в командной строке можно указать имя другого файла конфигурации. Это позволяет использовать разные конфигурационные файлы для каждого режима сборки.

## **26.4.2. Сборка JavaScript-файлов**

Создадим в каталоге `C:\book\p4\src` два файла: `index.js` (листинг 26.5) и `test.js` (листинг 26.6).

### **Листинг 26.5. Содержимое файла `C:\book\p4\src\index.js`**

```
import Test from './test';

var x = 10;
console.log(x);
var obj = new Test('Сообщение');
console.log(obj.getMsg());
```

### **Листинг 26.6. Содержимое файла `C:\book\p4\src\test.js`**

```
export default class Test {
 constructor(msg) {
 this.msg = msg;
 }
 getMsg() {
 return this.msg;
 }
}
```

Как видно из кода программы, для подключения модулей и экспорта идентификаторов можно использовать синтаксис модулей ECMAScript (см. *разд. 12.2*).

Далее в каталоге `C:\book\p4\dist` создаем страницу `test.html` (листинг 26.7), в которой подключим собранный файл `index.bundle.js`.

**Листинг 26.7. Содержимое файла `C:\book\p4\dist\test.html`**

```
<!doctype html>
<html lang="ru">
<head>
 <meta charset="utf-8">
 <title>Заголовок страницы</title>
</head>
<body>
 <h1>Заголовок</h1>
 <script src="index.bundle.js"></script>
</body>
</html>
```

Запустим скрипт `dev-pack`:

```
C:\book\p4>npm run dev-pack
```

```
> p4@1.0.0 dev-pack C:\book\p4
> webpack --mode development
```

```
asset index.bundle.js 4.29 KiB [emitted] (name: main)
runtime modules 695 bytes 3 modules
cacheable modules 248 bytes
 ./src/index.js 127 bytes [built] [code generated]
 ./src/test.js 121 bytes [built] [code generated]
webpack 5.38.1 compiled successfully in 159 ms
```

В результате будет создан файл `index.bundle.js` без сжатия, содержащий код из файлов `index.js` и `test.js`, а также большой объем вспомогательного кода. Откроем в веб-браузере страницу `test.html` и убедимся, что в консоли выводятся следующие сообщения:

```
10
```

Сообщение

Если запустить скрипт `build-pack`, файл `index.bundle.js` будет сжат.

### 26.4.3. Несколько точек входа

В приложении может быть несколько точек входа (например, две: основной скрипт и какая-либо JavaScript-библиотека). Тогда значение свойства `entry` в файле `webpack.config.js` может записываться в виде объекта, каждое свойство которого задает одну из точек входа. Имена этих свойств могут быть произвольными. Пример:



```
entry: {
 library: './src/library.js'.
 main: './src/index.js'
}
```

Содержимое файла `library.js` приведено в листинге 26.8.

#### Листинг 26.8. Содержимое файла `C:\book\p4\src\library.js`

```
function sum(x, y) {
 return x + y;
}
console.log(sum(10, 20));
```

В качестве значения свойства `output.filename` следует записать строковый шаблон для формирования имен результирующих файлов. В этих шаблонах можно использовать следующие символы-заполнители: `[name]` — имя свойства объекта, записанного в свойстве `entry` в файле `webpack.config.js`, `[contenthash]` — хеш, вычисленный на основе содержимого файла, `[hash]` — хеш на основе имени файла, `[id]` — внутренний идентификатор файла и др. Полный список заполнителей приведен в документации. Пример:

```
output: {
 filename: '[name].bundle.js',
 path: path.resolve(__dirname, 'dist')
}
```

В этом случае в каталоге `c:\book\p4\dist` будут созданы файлы скриптов `library.bundle.js` и `main.bundle.js`, которые останутся подключить ко всем страницам разрабатываемого сайта:

```
<script src="library.bundle.js"></script>
<script src="main.bundle.js"></script>
```

В состав имени результирующего файла можно включить хеш, вычисленный на основе его содержимого. Для этого достаточно изменить значение свойства `output` следующим образом:

```
output: {
 filename: '[name].[contenthash].js',
 path: path.resolve(__dirname, 'dist')
}
```

После сборки в каталоге `c:\book\p4\dist` будут созданы файлы примерно с такими именами:

```
library.b780ab64b52e2347af80.js
main.9fded3840c41674cc630.js
```

Причем после изменения содержимого исходного файла имя результирующего файла изменится (поскольку изменится хеш, вычисленный на основе его содержи-

мого). Это позволяет избежать ситуации, когда вместо новой редакции скрипта используется его старая редакция, сохраненная в кэше (так называемое "застывание в кэше").

Однако при разработке сайта формировать у файлов имена, содержащие хеш, неудобно, поскольку после каждой сборки проекта придется вручную исправлять имена этих файлов в HTML-коде всех страниц сайта (впрочем, в этом могут помочь средства массовой замены текста, встроенные в VS Code). Поэтому включать хеши в имена файлов рекомендуется лишь по окончании разработки сайта, при подготовке его эксплуатационной редакции.

#### 26.4.4. Очистка содержимого каталога

Еще одна проблема, возникающая при формировании имен файлов на основе хешей их содержимого: постепенное "засорение" каталога `c:\book\p4\dist` старыми версиями файлов со скриптами. Автоматически очищать этот каталог может пакет `clean-webpack-plugin`, устанавливаемый следующей командой:

```
C:\book\p4>npm install clean-webpack-plugin --save-dev
```

В файле `package.json` в секцию `devDependencies` будет добавлена строка:

```
"clean-webpack-plugin": "*",
```

В конфигурационном файле `webpack.config.js` подключаем модуль `clean-webpack-plugin`:

```
const { CleanWebpackPlugin } = require('clean-webpack-plugin');
```

Затем в разделе `plugins` создаем объект класса `CleanWebpackPlugin`:

```
new CleanWebpackPlugin({ cleanOnceBeforeBuildPatterns: ['*.js'] })
```

Настройка `cleanOnceBeforeBuildPatterns` задает массив шаблонов, которым должны удовлетворять имена файлов, подлежащих удалению. Файлы с именами, не удовлетворяющими ни одному из перечисленных в массиве шаблонов, не будут удалены. В нашем случае удаляться будут лишь JavaScript-файлы. Описание всех настроек пакета можно найти в документации.

Полное содержимое файла `webpack.config.js` приведено в листинге 26.9.

#### Листинг 26.9. Содержимое файла `C:\book\p4\webpack.config.js`

```
const path = require('path');
const { CleanWebpackPlugin } = require('clean-webpack-plugin');

module.exports = {
 entry: {
 main: './src/index.js',
 script: './src/script.js'
 },
```

```

output: {
 filename: '[name].[contenthash].js',
 path: path.resolve(__dirname, 'dist')
},
plugins: [
 new CleanWebpackPlugin({ cleanOnceBeforeBuildPatterns: ['*.js'] })
],
module: {}
};

```

Очистка каталога назначения от указанных файлов будет выполняться каждый раз при сборке проекта. Только будьте внимательны: не удалите случайно какие-либо нужные файлы.

### 26.4.5. Подключение файлов в формате JSON

Файлы в формате JSON можно подключать точно так же, как JavaScript-файлы.

Создадим в каталоге C:\book\p4\src файл data.json со следующим содержимым:

```

{
 "a": 20,
 "b": 30
}

```

Подключим этот файл в файле C:\book\p4\src\index.js:

```

import json from './data';

function sum(x, y) {
 return x + y;
}

console.log(sum(json.a, json.b));

```

Выполнив сборку проекта и открыв в веб-браузере страницу test.html, мы убедимся, что скрипт работает.

### 26.4.6. Подключение библиотек из каталога *node\_modules*

Если в инструкции `import` указано имя пакета без пути, то поиск пакета будет выполнен в каталоге `node_modules`. Установим библиотеку jQuery:

```
C:\book\p3>npm install jquery --save
```

В раздел `dependencies` файла `package.json` будет добавлена следующая строка:

```
"jquery": "^3.6.0"
```

Изменим содержимое файла C:\book\p4\src\index.js следующим образом:

```

import * as $ from 'jquery';

$('#div1').html('Сообщение');

```

После сборки код библиотеки jQuery будет добавлен в файл `main.bundle.js`. Если требуется, чтобы библиотечный код сохранялся в отдельном файле, то в конфигурационный файл `webpack.config.js` нужно добавить такие строки:

```
module: {
 . . .
 optimization: {
 splitChunks: {
 chunks: 'all',
 name: 'vendor'
 }
 }
}
```

Настройка `optimization.splitChunks.chunks` указывает, код каких библиотек следует сохранять в отдельном файле. Значение `all` этой настройки указывает Webpack вынести в отдельный файл код всех библиотек. Настройка `optimization.splitChunks.name` задает имя файла, в котором будет сохранен библиотечный код. К этому файлу будет автоматически добавлен суффикс `bundle.js`.

В результате код всех библиотек будет сохранен в файле `C:\book\p4\dist\vendor.bundle.js`. Этот файл следует подключить ко всем страницам разрабатываемого сайта:

```
<script src="vendor.bundle.js"></script>
<script src="main.bundle.js"></script>
```

## 26.4.7. Пакет babel-loader

Пакет `babel-loader` транслирует JavaScript-код нового стандарта ES6 в старый, "понимаемый" всеми присутствующими на рынке браузерами. Установим нужные пакеты:

```
C:\book\p4>npm install babel-loader @babel/core --save-dev
```

Дополнительно необходимо установить еще один пакет:

```
C:\book\p4>npm install @babel/preset-env --save-dev
```

В файле `package.json` в секцию `devDependencies` будут добавлены следующие строки:

```
"@babel/core": "^7.14.3",
"@babel/preset-env": "^7.14.4",
"babel-loader": "^8.2.2"
```

Содержимое конфигурационного файла `webpack.config.js` приведено в листинге 26.10.

### Листинг 26.10. Содержимое файла `C:\book\p4\webpack.config.js`

```
const path = require('path');

module.exports = {
```

```

entry: {
 script: './src/script.js'
},
output: {
 filename: '[name].[contenthash].js',
 path: path.resolve(__dirname, 'dist')
},
plugins: [],
module: {
 rules: [{
 test: /\.js$/i,
 exclude: /node_modules/,
 use: {
 loader: 'babel-loader',
 options: {
 presets: ['@babel/preset-env']
 }
 }
]
}
};

```

Изменим содержимое файла `C:\book\p4\src\index.js`, использовав стрелочную функцию и ключевое слово `const`, поддерживаемые только современными веб-браузерами:

```

const sum = (x, y) => x + y;
console.log(sum(10, 20));

```

Убедимся в наличии файла `.browserslistrc` (см. *разд. 26.1*) в корне пакета и выполним сборку. После сборки получим следующий код, поддерживаемый практически всеми веб-браузерами:

```

var sum = function sum(x, y) {
 return x + y;
};
console.log(sum(10, 20));

```

#### **ПРИМЕЧАНИЕ**

Некоторые возможности, предоставляемые Webpack, требуют установки дополнительных программ. Обратите внимание на текст сообщения об ошибке — обычно там указывается название необходимой программы.

## **26.4.8. Сборка SCSS-файлов**

По умолчанию Webpack работает только с JavaScript и JSON. Дополнительные возможности доступны с помощью отдельных программ. Установим программы, которые позволят выполнить трансляцию и сборку SCSS-файлов:

```

C:\book\p4>npm install style-loader css-loader sass-loader --save-dev
C:\book\p4>npm install mini-css-extract-plugin --save-dev

```

Также должен быть установлен пакет `node-sass` (см. *разд. 26.2.3*).

В файле `package.json` в секцию `devDependencies` будут добавлены следующие строки:

```
"css-loader": "^5.2.6",
"mini-css-extract-plugin": "^1.6.0",
"sass-loader": "^12.0.0",
"style-loader": "^2.0.0"
```

В каталоге `C:\book\p4\scss` создадим файл `main.scss` с содержимым из листинга 26.3. Затем подключим этот файл внутри файла `C:\book\p4\src\index.js`:

```
import '../scss/main.scss';

console.log('Сообщение');
```

Содержимое конфигурационного файла `webpack.config.js` приведено в листинге 26.11.

#### Листинг 26.11. Содержимое файла `C:\book\p4\webpack.config.js`

```
const path = require('path');
const MiniCssExtractPlugin = require('mini-css-extract-plugin');

module.exports = {
 entry: {
 main: './src/index.js',
 library: './src/library.js'
 },
 output: {
 filename: '[name].bundle.js',
 path: path.resolve(__dirname, 'dist')
 },
 devtool: 'source-map',
 plugins: [
 new MiniCssExtractPlugin({
 filename: '[name].bundle.css'
 })
],
 module: {
 rules: [{
 test: /\.s[ca]ss$/i,
 use: [MiniCssExtractPlugin.loader,
 'css-loader', {
 loader: 'sass-loader',
 options: {
 implementation: require('node-sass')
 }
 }
]
 }
]
};
```

Обратите внимание на порядок следования загрузчиков в свойстве `use`. Список просматривается справа налево. Вначале запускается `sass-loader`, затем — `css-loader` и последним — `MiniCssExtractPlugin.loader`. Последний загрузчик записывает результат в отдельный CSS-файл. Вместо этого загрузчика можно использовать `style-loader`:

```
use: ['style-loader', 'css-loader', 'sass-loader']
```

Тогда CSS-код будет вставлен в динамически сгенерированный тег `<style>` в теге `<head>`. Мы же создаем отдельный файл. Кроме того, создается еще и файл с расширением `css.map` благодаря инструкции:

```
devtool: 'source-map'
```

После сборки мы получим следующие файлы:

```
main.bundle.js
main.bundle.js.map
main.bundle.css
main.bundle.css.map
```

Благодаря наличию файлов с расширением `map` веб-браузер будет показывать номера строк в исходном коде, а не в сгенерированном. Это очень удобно на этапе разработки.

По умолчанию сборка выполняется с помощью пакета `sass`. Чтобы использовать `node-sass`, мы указываем это явным образом с помощью свойства `implementation`:

```
{
 loader: 'sass-loader',
 options: {
 implementation: require('node-sass')
 }
}
```

## 26.4.9. Подключение изображений

Webpack также позволяет подключать к модулям изображения и другие файлы. Для этого нужно установить пакет `file-loader`:

```
C:\book\p4>npm install file-loader --save-dev
```

В файле `package.json` в секцию `devDependencies` будет добавлена строка:

```
"file-loader": "^6.2.0"
```

Изменим содержимое файла `C:\book\p4\src\index.js` следующим образом:

```
import name from './photo.jpg';

console.log(name);
```

В тот же каталог поместим файл `photo.jpg`.

Содержимое конфигурационного файла `webpack.config.js` приведено в листинге 26.12.

**Листинг 26.12. Содержимое файла C:\book\p4\webpack.config.js**

```
const path = require('path');

module.exports = {
 entry: {
 main: './src/index.js',
 library: './src/library.js'
 },
 output: {
 filename: '[name].bundle.js',
 path: path.resolve(__dirname, 'dist')
 },
 plugins: [],
 module: {
 rules: [{
 test: /\. (png|jpg|gif)$/i,
 use: ['file-loader']
 }]
 }
};
```

Запускаем сборку с помощью команды:

```
C:\book\p4>npm run dev-pack
```

В итоге в каталоге C:\book\p4\dist будет создан файл с изображением, имеющий имя вида 81d4a9aa698bfc58af44a981ff042987.jpg. Получить путь к этому файлу внутри файла C:\book\p4\src\index.js позволит переменная `name`. Вот результат в консоли:

```
file:///C:/book/p4/dist/81d4a9aa698bfc58af44a981ff042987.jpg
```

## 26.4.10. Копирование файлов

Иногда бывает необходимо просто скопировать файл из каталога с исходным кодом в каталог с собранным сайтом. Выполнить такое копирование позволяет пакет `copy-webpack-plugin`. Установим его с помощью команды:

```
C:\book\p4>npm install copy-webpack-plugin --save-dev
```

В файле `package.json` в секцию `devDependencies` будет добавлена строка:

```
"copy-webpack-plugin": "^9.0.0"
```

Изменим содержимое файла C:\book\p4\src\index.js следующим образом:

```
console.log('Сообщение');
```

В тот же каталог поместим файл `photo.jpg`.

Содержимое конфигурационного файла `webpack.config.js` приведено в листинге 26.13.



**Листинг 26.13. Содержимое файла C:\book\p4\webpack.config.js**

```
const path = require('path');
const CopyWebpackPlugin = require('copy-webpack-plugin');

module.exports = {
 entry: {
 main: './src/index.js',
 library: './src/library.js'
 },
 output: {
 filename: '[name].bundle.js',
 path: path.resolve(__dirname, 'dist')
 },
 plugins: [
 new CopyWebpackPlugin({
 patterns: [{
 from: path.resolve(__dirname, 'src/photo.jpg'),
 to: path.resolve(__dirname, 'dist')
 }]
 })
],
 module: {}
};
```

В приведенном примере мы просто копируем файл `photo.jpg` из каталога `src` в каталог `dist`. Для этого конструктору класса `CopyWebpackPlugin` передаем объект со свойством `patterns`. Значением свойства является массив объектов, имеющих свойства `from` (путь к одному из копируемых файлов) и `to` (путь к каталогу назначения).

# Заключение

Вот и закончилось наше путешествие в мир языка JavaScript и программной платформы Node.js. В данной книге описаны лишь основы этих технологий. А здесь мы расскажем, где найти дополнительную информацию.

Самым важным источником информации является официальный сайт Node.js: <https://nodejs.org/ru/>. На этом сайте вы найдете новости, а также ссылки на другие интернет-ресурсы, посвященные Node.js и JavaScript. Регулярно посещайте страницу <https://nodejs.org/ru/download/>, на которой доступна для загрузки самая последняя версия Node.js.

На странице <https://tc39.es/ecma262/> расположена спецификация языка JavaScript (ECMAScript). Она обновляется очень часто, поскольку язык постоянно совершенствуется, обзаводится новыми языковыми конструкциями, операторами, функциями и т. д. Регулярно посещайте эту страницу — и вы будете в курсе самых "свежих" нововведений.

На страницах <https://ru.wikipedia.org/wiki/JavaScript> и <https://ru.wikipedia.org/wiki/ECMAScript> вы найдете историю развития JavaScript, а также действующие ссылки на все другие интернет-ресурсы, посвященные этому языку.

Если в процессе изучения JavaScript у вас возникнет какое-либо недопонимание, то не следует забывать, что Интернет предоставляет множество ответов на самые разнообразные вопросы. Достаточно в строке запроса поискового портала (например, <https://www.google.com/>) набрать свой вопрос. Наверняка уже кто-то сталкивался с подобной проблемой и описал ее решение на каком-либо сайте.

Библиотека Electron (<https://www.electronjs.org/>) предназначена для разработки оконных приложений на JavaScript. Пользовательский интерфейс таких приложений пишется на языках HTML и CSS, хорошо знакомых веб-разработчикам.

Программная платформа Deno — прямой конкурент Node.js. Ее основные особенности: поддержка языка программирования TypeScript (наряду с JavaScript), повышенная безопасность (так, изначально доступ к файловой системе для всех программ запрещен, и чтобы разрешить его, следует при запуске программы указать особый командный ключ) и поддержка только модулей стандарта ECMAScript. Найти описание этой платформы можно по интернет-адресу <https://deno.land/>.

Свои замечания и пожелания вы можете оставить на странице книги на сайте издательства "БХВ": <https://bhv.ru/>. Все замеченные опечатки и неточности просим присылать на E-mail [mail@bhv.ru](mailto:mail@bhv.ru) — только не забудьте указать название книги и имя автора.

# ПРИЛОЖЕНИЕ

## Описание электронного архива

Электронный архив к книге выложен на FTP-сервер издательства по адресу: **ftp://ftp.bhv.ru/9785977568470.zip**. Ссылка доступна и со страницы книги на сайте **https://bhv.ru**.

Содержимое архива представлено в табл. П.1.

*Таблица П.1. Содержимое электронного архива*

| Файл         | Описание                                          |
|--------------|---------------------------------------------------|
| Listings.doc | Все пронумерованные листинги, приведенные в книге |
| Readme.txt   | Описание электронного архива                      |

# Предметный указатель

## @

@babel-core 743  
@babel-preset-env 743

## —

\_\_dirname 355, 462  
\_\_filename 347, 355, 462  
\_\_proto\_\_ 276, 302

## A

abort() 676, 712  
AbortController 712  
AbortError 712  
abs() 165  
accept 641  
access() 543  
Access-Control-Allow-Credentials 726  
Access-Control-Allow-Headers 726  
Access-Control-Allow-Methods 726  
Access-Control-Allow-Origin 725, 726  
Access-Control-Max-Age 726  
Access-Control-Request-Headers 725  
Access-Control-Request-Method 725  
accessSync() 542  
accuracy 613  
acos() 167  
acosh() 167  
action 634, 635  
activeElement 578, 589  
add() 197, 598, 645  
addColorStop() 662, 663  
addEventListener() 618  
addListener() 437  
addRange() 600

AggregationError 427  
AJAX 671  
alert() 576, 585  
All Autocomplete 86  
all() 425  
alloc() 443  
allocUnsafe() 444  
allocUnsafeSlow() 445  
allSettled() 428  
alt 649  
altitude 613  
altitudeAccuracy 613  
altKey 631  
anchorNode 599  
anchorOffset 599  
any() 426  
append() 706, 713, 721  
appendChild() 592  
appendFile() 475, 478, 499  
appendFileSync() 472  
apply() 254, 273, 305, 335, 336  
arc() 660  
arcTo() 660  
args 399  
arguments 255, 306, 316  
argv 115  
argv0 115  
Array 175  
arrayBuffer() 711, 721  
arrowParens 412  
as 359, 367, 370  
asin() 167  
asinh() 167  
assert() 396  
assign() 294, 296, 588  
async 429, 571  
asyncIterator 433, 553  
atan() 167

atanh() 167  
 atime 541  
 atimeMs 541  
 atimeNs 541  
 attributes 595  
 author 100  
 Auto Close Tag 23  
 Auto Complete Tag 23  
 Auto Rename Tag 23  
 autocomplete 635, 638  
 AutoFileName 86  
 autofocus 636  
 autoplay 611  
 autoprefixer 731  
 availHeight 587  
 availWidth 587  
 await 429

## B

babel-loader 743  
 back() 588  
 badInput 636  
 basename() 469  
 beginPath() 659  
 bezierCurveTo() 661  
 bigint 123, 159, 172  
 BigInt() 173  
 BigIntStats 539  
 bind() 255, 273  
 birthtime 540  
 birthtimeMs 540  
 birthtimeNs 540  
 blob() 711, 720  
 blur() 637  
 body 578, 589, 708  
 bodyUsed 710, 719  
 boolean 123  
 Boolean 128, 139  
 Bootstrap 101  
 Bootstrap 5 Snippets 87  
 Bootstrap v4 Snippets 87  
 Bracket Pair Colorizer 86  
 bracketSpacing 411  
 break 158  
 browserslistrc 730  
 bubbles 624  
 buffer 460  
 Buffer 443  
 builtinModules 346, 360  
 button 629  
 byteLength() 448  
 byteOffset 460

## C

cache 708  
 call() 253, 273, 317  
 callee 406  
 cancelable 621  
 canPlayType() 612  
 canvas 653  
 CanvasGradient 662, 663  
 CanvasRenderingContext2D 653  
 caption 593  
 case 150  
 catch 390  
 catch() 424  
 cbrt() 165  
 ceil() 166  
 cellIndex 593  
 cells 593  
 charAt() 207  
 charCodeAt() 207  
 checked 644  
 checkValidity() 635  
 child\_process 354  
 childNodes 591  
 children 358  
 chmodSync() 544  
 class 307, 308, 597  
 classList 595, 597  
 className 595, 597  
 clean-css-cli 732  
 clean-webpack-plugin 741  
 clear() 115, 195, 198  
 clearImmediate() 246  
 clearInterval() 245  
 clearRect() 656  
 clearTimeout() 244  
 clearWatch() 616  
 clientX 629  
 clientY 629  
 clip() 661  
 clone() 710, 720  
 cloneContents() 602  
 cloneNode() 592  
 cloneRange() 602  
 close 519, 522, 557  
 close() 486, 488, 554, 557  
 closePath() 659  
 closeSync() 485, 554  
 code 614, 630  
 collapse() 599, 602  
 collapsed 602  
 collapseToEnd() 600

collapseToStart() 599  
colorDepth 587  
COMMENT\_NODE 591  
commonAncestorContainer 602  
CommonJS 345  
compare() 456  
compareBoundaryPoints() 603  
complete 610  
concat() 182, 207, 453  
configurable 279  
confirm() 577, 585  
console 109, 112, 396  
const 128  
construct() 334, 336  
constructor 303, 314  
constructor() 308  
contains() 598  
containsNode() 600  
continue 157  
controls 611  
cookie 589, 604  
Cookie 604  
cookieEnabled 587, 604  
coords 613  
copy() 451  
copyFile() 530, 531  
copyFileSync() 529  
copy-webpack-plugin 747  
copyWithin() 178  
CORS 724  
cos() 166  
cosh() 167  
create() 277, 278  
createCaption() 594  
createElement() 592  
createImageData() 668  
createLinearGradient() 662  
createObjectURL() 720  
createPattern() 664  
createRadialGradient() 663  
createRange() 601  
createReadStream() 520  
createServer() 560  
createTBody() 594  
createTextNode() 592  
createTFoot() 594  
createTHead() 594  
createWriteStream() 516  
credentials 708, 726  
CSS Peek 87  
css-loader 744  
ctime 540

ctimeMs 540  
ctimeNs 540  
ctrlKey 631  
currency 169  
currencyDisplay 169  
currentSrc 611  
currentTarget 620  
currentTime 611  
customError 636  
cwd() 463

## D

data 521, 522, 668, 690  
Date 238  
decode() 461  
decodeURI() 213  
decodeURIComponent() 213  
default 149, 368, 380  
defaultChecked 644  
defaultMaxListeners 439  
defaultPrevented 622  
defaultSelected 646  
defaultValue 638  
defer 571, 573  
defineProperties() 283, 330  
defineProperty() 275, 282, 283, 330, 331, 338  
delete 184, 296, 331  
delete() 195, 198, 706, 715, 723  
deleteCaption() 594  
deleteCell() 594  
deleteContents() 602  
deleteFromDocument() 600  
deleteProperty() 297, 331, 332, 338  
deleteRow() 594  
deleteTFoot() 594  
deleteTHead() 594  
delimiter 346, 359, 466  
deltaMode 629  
deltaX 629  
deltaY 629  
deltaZ 629  
dependencies 101, 103  
description 100  
detail 629  
devDependencies 101, 103  
Dir 551, 552  
Dirent 554  
dirname() 469  
disabled 636, 646  
do...while 154  
document 585, 588

DOCUMENT\_NODE 591  
 documentElement 589  
 DocumentFragment 602  
 DOM 584  
 DOMTokenList 597, 598  
 done 262, 318  
 dotAll 225  
 drain 519  
 drawImage() 658  
 duration 611

## E

E 164  
 ECMAScript 15, 358  
 ELEMENT\_NODE 591  
 elementFromPoint() 590  
 elements 634  
 emit() 436  
 Emmet 42, 52, 75  
 empty() 600  
 enableHighAccuracy 615  
 encode() 461  
 encodeURI() 213  
 encodeURIComponent() 213  
 encodingExists() 461  
 enctype 634, 635  
 end 522  
 end() 518, 564  
 END\_TO\_END 603  
 END\_TO\_START 603  
 endContainer 601  
 ended 612  
 endOffset 601  
 endOfLine 410  
 endsWith() 211  
 entries() 191, 196, 199, 288, 299, 456, 704, 714, 722  
 enumerable 279  
 EPSILON 162  
 equals() 457  
 error 519, 522  
 Error 392  
 error() 114, 390  
 errors 427  
 ESLint 415  
 eslint-config-airbnb-base 415  
 eslint-config-prettier 415  
 eslint-plugin-import 415  
 eslint-plugin-prettier 415  
 eval() 214, 344, 692  
 EvalError 393

event 624  
 Event 620, 621, 629  
 EventEmitter 436  
 eventNames() 439  
 eventPhase 625  
 events 436, 442  
 every() 189  
 exec() 222  
 execPath 115  
 execSync() 354  
 existsSync() 544  
 exit() 117, 391  
 exitCode 117  
 exp() 165  
 expires 605  
 expm1() 165  
 export 364, 369  
 export default 367  
 exports 356, 357, 358, 377, 378  
 extend() 600  
 extends 309  
 extname() 469  
 extractContents() 602

## F

false 123, 137, 143  
 fd 514  
 Fetch API 698  
 fetch() 698  
 File 642  
 FileList 641  
 file-loader 746  
 filename 358  
 filepath 410  
 files 641  
 fill() 178, 445, 448, 659  
 fillRect() 656  
 fillStyle 654  
 fillText() 657  
 filter() 190  
 finally 390  
 finally() 425  
 find() 190  
 findIndex() 190  
 finish 519  
 firstChild 591, 690  
 flags 226  
 flat() 180  
 flatMap() 188  
 floor() 166  
 focus() 637

focusNode 599  
focusOffset 599  
font 657  
for 151, 182, 454  
for await...of 431  
for() 291  
for...in 155, 193, 289  
for...of 157, 183, 192, 196, 199, 317, 454  
forEach() 182, 196, 199, 455, 705, 715, 723  
form 634, 636  
format() 170, 468  
FormData 712, 713, 715  
formData() 711, 720  
forms 585, 590, 634  
forward() 588  
frames 584, 586  
freeze() 284  
from() 176, 181, 200, 212, 316, 445–447, 451, 459  
fromCharCode() 203  
fromEntries() 299  
fs 470, 480, 485, 510, 516, 520, 526, 529, 532, 535, 544, 545, 548, 551, 556  
fstat() 537  
fstatSync() 536  
FSWatcher 557  
ftruncate() 512  
ftruncateSync() 511  
fulfilled 423  
function 123, 249  
Function 253

## G

geolocation 613  
Geolocation 613  
GeolocationCoordinates 613  
GeolocationPosition 613  
GeolocationPositionError 614  
get 275, 280  
get() 194, 325, 327, 337, 704, 713, 722  
getAll() 704, 714  
getAllResponseHeaders() 676  
getAttribute() 595  
getComputedStyle() 597  
getContext() 653  
getCurrentPosition() 613–615  
getDate() 241  
getDay() 242  
getElementById() 572, 578, 589  
getElementsByClassName() 578, 590, 596  
getElementsByName() 578, 589

getElementsByTagName() 578, 589, 596, 690  
getFullYear() 242  
getHeader() 564  
getHeaderNames() 564  
getHeaders() 564  
getHours() 242  
getImageData() 667  
getItem() 608  
getLineDash() 655  
getMaxListeners() 439  
getMilliseconds() 242  
getMinutes() 242  
getMonth() 241  
getOwnPropertyDescriptor() 281, 329, 330, 339  
getOwnPropertyDescriptors() 281  
getOwnPropertyNames() 288  
getOwnPropertySymbols() 292  
getPrototypeOf() 277, 333, 334, 339  
getRangeAt() 600, 601  
getResponseHeader() 676  
getSeconds() 242  
getSelection() 599  
getter 275, 280  
getTime() 241  
getTimezoneOffset() 243  
getUTCDate() 241  
getUTCDay() 242  
getUTCFullYear() 242  
getUTCHours() 242  
getUTCMilliseconds() 242  
getUTCMinutes() 242  
getUTCMonth() 241  
getUTCSeconds() 242  
global 225, 385  
globalAlpha 666  
globalCompositeOperation 665  
globalPaths 354  
globalThis 386  
go() 588

## H

has() 194, 198, 327, 337, 705, 715, 723  
hasAttribute() 596  
hasChildNodes() 593  
hash 588, 703  
hasHeader() 564  
hasIndices 225  
hasOwnProperty() 156, 290  
head 589  
header() 674  
headers 562, 707, 719



Headers 721  
heading 613  
height 587, 611, 653  
history 584, 588  
host 588, 702  
hostname 587, 702  
href 587, 702  
HTML CSS Support 86  
http 560  
httpVersion 563  
httpVersionMajor 563  
httpVersionMinor 563  
hypot() 165

## I

iconv-lite 461  
id 358, 572, 595  
if...else 141  
ignoreCase 225  
ImageData 667, 668  
images 585, 590  
import 359, 369, 371, 380  
import() 372  
import.meta.url 361, 462  
imports 383  
in 290, 327  
includes() 189, 210, 458  
IncomingMessage 561, 562  
indeterminate 644  
index 646  
indexOf() 189, 210, 457  
Infinity 171  
info() 113  
innerHeight 585  
innerHTML 572, 595  
innerWidth 585  
insertBefore() 592  
insertCell() 594  
insertNode() 603  
insertRow() 594  
instanceof 305, 315  
Int16Array 443  
Int32Array 443  
Int8Array 443  
integrity 709  
Intl 170  
is() 137, 172, 293  
isAbsolute() 466  
isArray() 176  
isBuffer() 447  
isCollapsed 599

isDirectory() 540, 554  
isEncoding() 460  
isExtensible() 287, 332, 333, 340  
isFile() 540, 554  
isFinite() 163, 172  
isFrozen() 285  
isInteger() 163  
isNaN() 143, 164, 172  
isPointInPath() 662  
isPrototypeOf() 277, 304  
isSafeInteger() 163  
isSealed() 286  
item() 598  
iterator 318

## J

JavaScript 15  
join() 187, 464, 467  
jQuery 101, 582, 742  
jQuery Code Snippets 87  
JSON 340, 692  
json() 711, 720  
json\_decode() 695  
json\_encode() 695

## K

keepalive 712  
key 630  
keyFor() 291  
keys() 191, 195, 199, 287, 455, 704, 714, 722  
keywords 100

## L

language 586  
lastChild 591  
lastIndex 224  
lastIndexOf() 189, 210, 458  
lastMatch 235  
lastModified 589, 642  
latest 103  
latitude 613  
length 179, 207, 447, 586, 591, 598, 635, 645, 668  
lengthComputable 678  
let 121, 122, 151  
license 100  
lineCap 654  
lineDashOffset 655  
lineJoin 655

lineTo() 660  
lineWidth 654  
links 585, 591  
listen() 561  
listenerCount() 439  
listeners() 439  
Live Sass Compiler 87  
Live Server 41  
LN10 164  
LN2 164  
load() 612  
loaded 358, 678  
localeCompare() 210  
localStorage 608  
location 584, 587, 631  
log() 109, 112, 132, 165, 396  
log10() 165  
LOG10E 164  
log1p() 165  
log2() 165  
LOG2E 164  
longitude 613  
loop 611

## M

main 100, 364, 376  
Map 193  
map() 187  
match() 212, 216  
matchAll() 212, 217  
Math 164, 165, 166, 171  
max 640  
max() 165  
MAX\_SAFE\_INTEGER 161  
MAX\_VALUE 161  
maximumAge 615  
maximumFractionDigits 169  
maxLength 638  
measureText() 658  
message 392, 614  
metaKey 631  
method 562, 634, 635, 707  
min 640  
min() 165  
MIN\_SAFE\_INTEGER 161  
MIN\_VALUE 161  
mini-css-extract-plugin 744  
minimumFractionDigits 169  
minimumIntegerDigits 169  
minLength 638  
miterLimit 655

mkdir() 546  
mkdirSync() 545  
mkdtemp() 547, 548  
mkdtempSync() 547  
mode 540, 708  
module 346, 354, 356–358, 360  
module.exports 356  
move\_uploaded\_file() 718  
moveTo() 659  
mtime 541  
multiline 225  
multiple 641, 645  
muted 611

## N

name 100, 392, 399, 554, 586, 595, 634, 642  
NaN 125, 142, 161, 171  
naturalHeight 610  
naturalWidth 610  
navigator 584, 586  
NEGATIVE\_INFINITY 161, 171  
networkState 612  
new 300, 334  
new.target 301  
next 103  
next() 192, 262, 318  
nextSibling 592  
nextTick() 441  
node 380  
Node Package Manager 97  
Node.js  
  ♦ версия 93, 117  
  ♦ установка 88  
node\_modules 353, 364  
NODE\_PATH 354  
Nodemon 566  
nodeName 591  
node-sass 733  
nodeType 591  
nodeValue 591  
normalize() 466  
noscript 571  
Notepad++ 19  
noValidate 635  
now() 238  
NPM 97  
npm-run-all 737  
null 123  
number 123, 159  
Number 128, 160, 169, 171  
NumberFormat 170

## O

O\_APPEND 490  
 O\_CREAT 490  
 O\_EXCL 490  
 O\_RDONLY 490  
 O\_RDWR 490  
 O\_TRUNC 490  
 O\_WRONLY 490  
 object 123  
 Object 264, 298  
 of() 176  
 off() 440  
 offsetX 629  
 offsetY 629  
 ok 719  
 on() 437, 442  
 onabort 632, 677  
 onafterprint 626  
 onbeforeprint 626  
 onbeforeunload 626  
 onblur 637  
 oncanplay 631  
 oncanplaythrough 631  
 once() 438  
 onchange 638  
 onclick 627  
 oncontextmenu 628  
 ondblclick 627  
 ondurationchange 631  
 onemptied 632  
 onended 631  
 onerror 632, 678  
 onfocus 637  
 oninput 638  
 oninvalid 637  
 onkeydown 630  
 onkeypress 630  
 onkeyup 630  
 onLine 587  
 onload 626, 678  
 onloadeddata 631  
 onloadedmetadata 631  
 onloadend 678  
 onloadstart 631, 678  
 onmousedown 627  
 onmousemove 627  
 onmouseout 628  
 onmouseover 628  
 onmouseup 627  
 onpause 631  
 onplay 631  
 onplaying 632  
 onprogress 631, 678  
 onratechange 632  
 onreadystatechange 677  
 onreset 635  
 onresize 626  
 onscroll 626  
 onseeked 632  
 onseeking 632  
 onselect 638  
 onstalled 632  
 onsubmit 635  
 ontimeout 677  
 ontimeupdate 631  
 onunload 626  
 onvolumechange 631  
 onwaiting 632  
 onwheel 628  
 open 519, 522  
 open() 486, 487, 585, 675  
 opendir() 551, 552  
 opendirSync() 551  
 openSync() 485  
 options 645  
 origin 702  
 Origin 725, 726  
 os 346, 360, 463  
 outerHeight 585  
 outerWidth 585  
 outputCapture 400  
 overrideMimeType() 676  
 overrides 412  
 ownKeys() 289, 328, 338

## P

package.json 98  
 padEnd() 211  
 padStart() 211  
 pageX 629  
 pageXOffset 585  
 pageY 629  
 pageYOffset 585  
 parent 586  
 parentNode 591  
 parse() 239, 343, 468, 692  
 parseFloat() 126, 163, 168  
 parseInt() 126, 163, 167  
 parser 410  
 password 703  
 path 358, 462, 466, 552  
 PATH 93  
 pathname 588, 703  
 paths 358

pattern 638  
patternMismatch 636  
pause() 612  
paused 611  
pending 423  
PI 164  
pipe 523  
pipe() 523  
placeholder 638  
platform() 346  
play() 612  
playbackRate 612  
Polacode 87  
pop() 185  
port 588, 702  
PositionOption 615  
POSITIVE\_INFINITY 162, 171  
posix 463, 466  
postcss 731  
postcss-cli 731  
poster 611  
pow() 165  
preload 611  
prependListener() 438  
prependOnceListener() 438  
Prettier 407, 415  
preventDefault() 621  
preventExtensions() 287, 332, 333, 340  
previousSibling 591  
printWidth 410  
process 115, 117  
program 400  
Promise 421  
prompt() 577, 585  
propertyIsEnumerable() 156, 280, 289  
protocol 587, 702  
prototype 301  
Proxy 322  
push() 184  
putImageData() 669

## Q

quadraticCurveTo() 661  
querySelector() 579, 590, 596  
querySelectorAll() 579, 590, 596  
quoteProps 410

## R

race() 427  
random() 171

Range 601  
rangeCount 599  
RangeError 393  
rangeOverflow 636  
rangeUnderflow 636  
raw() 206  
rawHeaders 562  
rawListeners() 439  
read() 503, 506, 507, 521, 552, 553  
readable 521, 522  
readBigInt64BE() 450  
readBigInt64LE() 450  
readBigUInt64BE() 450  
readBigUInt64LE() 450  
readdir() 555, 556  
readdirSync() 555  
readDoubleBE() 450  
readDoubleLE() 450  
readFile() 481, 483, 508  
readFileSync() 480  
readFloatBE() 450  
readFloatLE() 450  
readInt16BE() 450  
readInt16LE() 450  
readInt32BE() 450  
readInt32LE() 450  
readInt8() 450  
readIntBE() 450  
readIntLE() 450  
readOnly 636  
ReadStream 520  
readSync() 500, 552  
readUInt16BE() 450  
readUInt16LE() 450  
readUInt32BE() 450  
readUInt32LE() 450  
readUInt8() 450  
readUIntBE() 450  
readUIntLE() 450  
readv() 505, 509  
readvSync() 502  
ready 519, 522  
readyState 589, 612, 677  
rect() 661  
redirect 708  
redirected 719  
reduce() 188  
reduceRight() 188  
ReferenceError 393  
referrer 589, 709  
referrerPolicy 709  
Reflect 336

RegExp 215, 222  
reject() 422, 423  
rejected 423  
relatedTarget 629  
relative() 468  
reload() 588  
remove() 598, 646  
removeAllListeners() 440  
removeAllRanges() 600  
removeAttribute() 596  
removeChild() 593  
removeEventListener() 619  
removeItem() 609  
removeListener() 440  
removeRange() 600  
rename() 527, 528  
renameSync() 526  
repeat 631  
repeat() 213  
replace() 212, 220, 588  
replaceAll() 212, 221  
replaceChild() 593  
request 399  
Request 707, 710  
require 380  
require() 345, 348, 356  
require.cache 350  
require.main 345  
required 636  
reset() 635  
resolve() 352, 422, 423, 463, 464, 467  
resolvedOptions() 170  
response 679, 680  
Response 718–720  
responseText 679, 680  
responseType 679  
responseXML 679  
REST 256, 257, 261  
restore() 664  
return 249  
reverse() 185  
revocable() 335  
rm() 534, 535, 549, 550  
rmdir() 550  
rmdirSync() 548  
rmSync() 532, 549  
rotate() 665  
round() 166  
rowIndex 593  
rows 593  
runtimeArgs 399  
runtimeExecutable 399

## S

sass 101  
sass-loader 744  
save() 664  
scale() 665  
screen 584, 587  
screenLeft 585  
screenTop 585  
screenX 585, 629  
screenY 585, 629  
script 570  
scripts 100, 104, 585, 591  
scrollBy() 585  
scrollIntoView() 596  
scrollTo() 585  
scrollX 585  
scrollY 585  
SCSS 733  
SCSS IntelliSense 87  
seal() 286  
search 588, 703  
search() 212, 216  
searchParams 703  
sectionRowIndex 593  
seeking 612  
select() 638  
selectAllChildren() 600  
selected 646  
selectedIndex 645  
selectedOptions 645  
selection 599  
selectionEnd 600, 638  
selectionStart 600, 638  
selectNode() 602  
selectNodeContents() 602  
self 586  
semi 410  
send() 676  
sep 359, 466  
Server 561  
ServerResponse 561, 563  
sessionStorage 608  
set 274, 280  
Set 197  
set() 194, 324, 327, 337, 705, 713, 721  
setAttribute() 596  
setCustomValidity() 651  
setEncoding() 521  
setEnd() 602  
setEndAfter() 602  
setEndBefore() 602

setHeader() 564  
setImmediate() 246, 441  
setInterval() 244  
setItem() 608  
setLineDash() 655  
setMaxListeners() 439  
setPosition() 599  
setPrototypeOf() 277, 333, 334, 339  
setRequestHeader() 676  
setSelectionRange() 601  
setStart() 602  
setStartAfter() 602  
setStartBefore() 602  
setter 274, 280  
setTime() 239  
setTimeout() 244  
shadowBlur 667  
shadowColor 667  
shadowOffsetX 667  
shadowOffsetY 667  
shift() 184  
shiftKey 631  
sign() 166  
signal 712  
sin() 166  
singleQuote 410  
sinh() 167  
size 194, 198, 540, 642, 645  
skipFiles 400  
slice() 181, 186, 209, 317, 451, 452  
some() 190  
sort() 185  
source 226  
speed 613  
spellcheck 642  
splice() 185  
split() 212, 221  
SPREAD 175, 178, 181, 182, 257, 293, 295,  
317, 459  
sqrt() 165  
SQRT1\_2 164  
SQRT2 164  
src 571, 611, 649  
stack 392  
START\_TO\_END 603  
START\_TO\_START 603  
startContainer 601  
startOffset 601  
startsWith() 211  
stat() 536–538  
static 313  
Stats 535–538

statSync() 535  
status 677, 719  
statusCode 564  
statusMessage 564  
statusText 677, 719  
StatWatcher 558  
stderr 114, 514, 525  
stdin 514  
stdout 112, 359, 398, 514, 525  
step 640  
stepDown() 640  
stepMismatch 636  
stepUp() 640  
sticky 225  
stopPropagation() 624  
Storage 608  
string 123  
String 128, 203  
stringify() 340  
stroke() 659  
strokeRect() 656  
strokeStyle 654  
strokeText() 657  
style 169, 596  
style-loader 744  
subarray() 452  
submit() 635  
substr() 209  
substring() 209  
super 310  
super() 310  
surroundContents() 603  
swap16() 453  
swap32() 454  
swap64() 454  
switch 148, 149  
symbol 123, 291  
Symbol 291  
SyntaxError 387, 393

## T

table() 113, 176  
tabWidth 410  
tagName 595  
tan() 166  
tanh() 167  
target 620, 635  
tBodies 593  
test() 222  
text 646  
text() 710, 720

TEXT\_NODE 591  
textAlign 657  
textBaseline 658  
textContent 595  
tfoot 593  
thead 593  
then() 423  
this 253–255, 272, 298, 300, 619  
throw 393  
time() 247  
timeEnd() 247  
timeLog() 247  
timeout 615, 677  
timestamp 620  
title 588  
tmpdir() 547  
toDate() 240  
toExponential() 162, 169  
toFixed() 162, 169  
toggle() 598  
toISOString() 240  
toJSON() 240, 341, 447  
toLocaleDateString() 240  
toLocaleLowerCase() 208  
toLocaleString() 163, 169, 174, 187, 240  
toLocaleTimeString() 241  
toLocaleUpperCase() 208  
toLowerCase() 208  
tooLong 636  
tooShort 636  
top 586  
toPrecision() 162, 169  
toPrimitive 446  
toString() 126, 162, 168, 174, 187, 203, 240,  
298, 306, 459, 599, 602, 703  
total 678  
toTimeString() 241  
toUpperCase() 208  
toUTCString() 240  
trailingComma 412  
transcode() 460  
translate() 665  
trim() 211  
trimLeft() 211  
trimRight() 211  
true 123, 137, 144  
trunc() 166  
truncate() 512, 513  
truncateSync() 510  
try...catch 389  
type 399, 570, 620, 636, 642, 645, 719  
TypeError 393

typeMismatch 636  
typeof 124  
TypeScript 103

## U

Uint16Array 443  
Uint32Array 443  
Uint8Array 443, 446  
undefined 124  
unicode 226  
unlink() 533, 534  
unlinkSync() 532  
unpipe 523  
unpipe() 523  
unshift() 184  
unwatchFile() 559  
upload 679  
URIError 393  
url 562, 719  
URL 589, 701, 702  
URLSearchParams 703  
URL-кодирование строки 213  
use strict 402  
useGrouping 170  
userAgent 586  
username 703  
useTabs 410  
UTC() 239  
utimesSync() 541

## V

valid 637  
validationMessage 636  
validity 636  
ValidityState 636  
value 262, 279, 318, 595, 638, 645, 646, 649  
valueMissing 636  
valueOf() 162, 203, 241, 307  
values() 191, 195, 199, 288, 455, 704, 714,  
722  
var 121, 568  
version 100, 117  
version() 346  
videoHeight 611  
videoWidth 611  
visibilityState 589  
Visual Studio Code 19  
volume 611  
VS Code 19

**W**

warn() 114  
watch() 556  
watchFile() 558  
watchPosition() 616  
Webpack 737  
webpack-cli 737  
while 153  
width 587, 611, 653  
willValidate 637  
win32 463, 466  
window 568, 585, 586  
withCredentials 679  
wrap 642  
writable 279, 518  
writableEnded 518, 564  
write() 114, 398, 449, 493, 496, 518, 563, 571, 590  
writeBigInt64BE() 449  
writeBigInt64LE() 449  
writeBigUInt64BE() 449  
writeBigUInt64LE() 449  
writeDoubleBE() 449  
writeDoubleLE() 449  
writeFile() 473, 477, 499  
writeFileSync() 470  
writeFloatBE() 449  
writeFloatLE() 449  
writeHead() 564

writeInt16BE() 449  
writeInt16LE() 449  
writeInt32BE() 449  
writeInt32LE() 449  
writeInt8() 449  
writeIntBE() 449  
writeIntLE() 449  
writeln() 590  
WriteStream 516  
writeSync() 491  
writeUInt16BE() 449  
writeUInt16LE() 449  
writeUInt32BE() 449  
writeUInt32LE() 449  
writeUInt8() 449  
writeUIntBE() 449  
writeUIntLE() 449  
writev() 495, 498  
writevSync() 492

**X**

x 629  
XMLHttpRequest 675, 677, 680

**Y**

y 629  
yield 262

**A**

Абсолютное значение 165  
Арккосинус 167  
Арсинус 167  
Арттангенс 167  
Асинхронность 421  
Асинхронный итератор 433  
Атрибут свойства 278

**Б**

Бесконечность 171  
Блок 130

**B**

Веб-браузер 568  
Веб-сервер 560  
Возведение в степень 134  
Время 238  
Всплывание события 624  
Выделение 599  
Вычитание 133

**Г**

Геолокация 613  
Градиент 662  
Группа 232



## Д

Дата 238  
Действие по умолчанию 621  
Декремент 134  
Деление 133

## З

Зависимость 101  
Замыкание 252, 259  
Запятая 136

## И

Именованная переменная 119  
Инверсия 135  
Инициализация 120  
Инкремент 134  
Инструменты разработчика 575  
Интерактивный режим 111  
Исключение 389  
◇ генерирование 393  
◇ обработка 389  
Итератор 317  
◇ асинхронный 433

## К

Квадратный корень 165  
Квантификатор 231  
Класс 299, 307  
◇ базовый 310  
◇ производный 310  
Кодировка 443, 460  
Командная строка 92  
Комментарии 109  
Конкатенация строк 206  
Консоль 92, 575  
Консольное приложение 94  
Константа 128  
Конструктор класса 300, 308  
Контекст рисования 653  
Косинус 166, 167  
Кросс-доменный запрос 724  
Кубический корень 165

## Л

Логарифм 165  
Лямбда-выражение 260

## М

Массив 175  
◇ ассоциативный 192, 193  
◇ добавление элемента 183  
◇ зубчатый 180  
◇ изменение элемента 177  
◇ инициализация 175  
◇ многомерный 179  
◇ объединение 182  
◇ перебор элементов 182  
◇ переворачивание 185  
◇ поиск элемента 189  
◇ получение части 186  
◇ получение элемента 177  
◇ преобразование в массив 187  
◇ преобразование в строку 187  
◇ размер 179  
◇ создание копии 181  
◇ сортировка 185  
◇ удаление элемента 184  
◇ фильтрация 190  
Математическая константа 164  
Метасимвол 228  
Метод  
◇ генератор 319  
◇ приватный 312  
◇ статический 307, 313  
Метод объекта 270  
Множество 197  
◇ объединение 200  
◇ пересечение 200  
◇ разница 201  
◇ симметрическая разность 201  
Модуль 345  
◇ CommonJS 345  
◇ ECMAScript 358  
Мультимедиа 611

## Н

Наследование 303, 309  
Неименованный блок 130

## О

Обработка исключений 389  
Обработка ошибок 387  
Обработчик события 435  
Обратная ссылка 235

Объединение множеств 200  
Объект 264  
◊ вложенный 267  
◊ массивоподобный 316  
Объектная модель документа 584  
Объектно-ориентированное  
программирование (ООП) 264  
Оператор 133  
◊ ? 268  
◊ ?. 271  
◊ ?: 147  
◊ break 158  
◊ continue 157  
◊ in 290  
◊ new 300  
◊ ветвления 141  
◊ выбора 148  
◊ запятая 136  
◊ математический 133  
◊ нулевого слияния 139  
◊ опциональной последовательности 268,  
271  
◊ побитовый 135  
◊ приоритет выполнения 140  
◊ присваивания 139  
◊ сравнения 137  
◊ условный 141  
Остаток от деления 133  
Отладка программы 396  
Отладчик 580  
Ошибка 387  
◊ времени выполнения 389  
◊ логическая 388

## П

Пакет 345  
Переменная 119  
◊ вывод значения 132  
◊ глобальная 129  
◊ именованная 119  
◊ инициализация 120  
◊ локальная 129  
◊ локальная внутри блока 129  
◊ область видимости 129  
◊ объявление 120  
◊ присваивание значения 120  
◊ проверка существования 124  
◊ уровня приложения 385  
Переопределение метода 310

Подкласс 310  
Пользовательские фрагменты кода 77  
Поток 516  
◊ ввода 516  
◊ вывода 516  
Потомок 591  
Пошаговое выполнение программы 401  
Права доступа 542  
Присваивание 120  
Прототип 276, 301  
Псевдомассив 316  
Путь 463

## Р

Разница множеств 201  
Распаковка объекта 268  
Регулярное выражение 215  
Рекурсия 263  
Репозиторий 100  
Родитель 591

## С

Свойство  
◊ приватное 312  
◊ публичное 311  
◊ статическое 307, 313  
Свойство объекта 265  
Сервер 560  
Симметрическая разница множеств 201  
Синус 166, 167  
Скрипт 104, 568  
◊ запуск 107  
Сложение 133  
Событие 435  
◊ обработка 435  
Специальный символ 204  
Строгий режим 402  
Строка 202  
◊ URL-кодирование 213  
◊ в обратных кавычках 205  
◊ длина 207  
◊ замена 210  
◊ изменение регистра 208  
◊ инициализация 202  
◊ конкатенация 206  
◊ неформатированная 206  
◊ повтор 213  
◊ поиск 210  
◊ получение фрагмента 209

Строка (*прод.*)

- ◇ преобразование в массив 212
- ◇ специальный символ 204
- ◇ сравнение 209
- ◇ шаблон строки 205
- Суперкласс 310

## Т

- Таймер 244, 579
- Тангенс 166, 167
- Текстура 664
- Текущий рабочий каталог 463
- Тип данных 123
- ◇ преобразование 125
- Точки останова 397

## У

- Узел 591
- ◇ дочерний 591
- ◇ родительский 591
- Умножение 133
- Унарный минус 134
- Унарный плюс 134
- Установка Node.js 88
- Установка пакетов 101

## Ф

- Фаза события 625
- Файл 462
- ◇ отправка 715
- ◇ режим открытия 489
- ◇ создание 35
- Факториал 263
- Форматирование кода 85, 394, 407
- Функция 249
- ◇ анонимная 258
- ◇ вложенная 252
- ◇ вызов 250
- ◇ генератор 261
- ◇ замыкание 252
- ◇ необязательный параметр 257
- ◇ передача параметров 256
- ◇ произвольное число параметров 255
- ◇ расположение 251

- ◇ рекурсия 263
- ◇ создание 249
- ◇ стрелочная 260
- ◇ тригонометрическая 166

## Х

- Холст 653
- Хранилище 607
- ◇ локальное 608
- ◇ сессионное 608

## Ц

- Цикл
- ◇ do...while 154
- ◇ for 151
- ◇ for await...of 431
- ◇ for...in 155
- ◇ for...of 157
- ◇ while 153
- Цикл обработки событий 436

## Ч

- Число 159
- ◇ bigint 159, 172
- ◇ number 159
- ◇ Number 160
- ◇ восьмеричное 159
- ◇ двоичное 159
- ◇ математическая константа 164
- ◇ ограничения точности 160
- ◇ округление 166
- ◇ преобразование 167, 168
- ◇ псевдослучайное 171
- ◇ указание значения 159
- ◇ шестнадцатеричное 159

## Ш

- Шаблон строки 205
- Шаблоны строк 132

## Э

- Экспонента 165