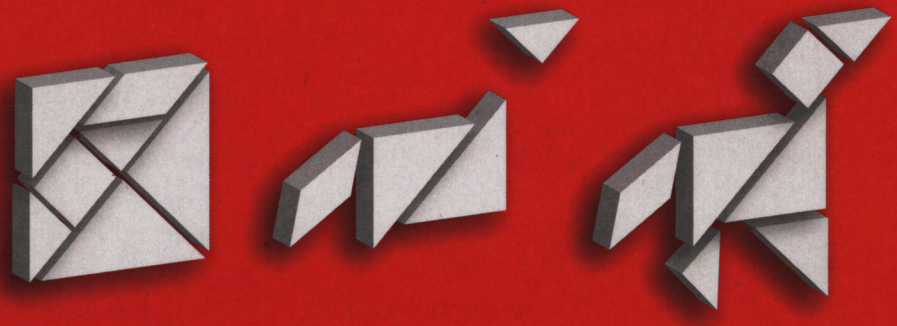
Владимир Дронов

JavaScript

20 уроков для начинающих



+ Материалы на www.bhv.ru

bhv®

Владимир Дронов

JavaScript

20 уроков для начинающих

Санкт-Петербург
«БХВ-Петербург»
2021

УДК 004.438JavaScript
ББК 32.973.26-018.1
Д75

Дронов В. А.

Д75 JavaScript: 20 уроков для начинающих. — СПб.: БХВ-Петербург, 2021. — 352 с.: ил. — (Для начинающих)

ISBN 978-5-9775-6589-9

В книге 20 иллюстрированных уроков, 40 практических упражнений на тему программирования веб-сценариев и более 70 заданий для самостоятельной работы. Изложены основы JavaScript: данные и операторы, выражения и управляющие конструкции, функции, классы, объекты и массивы, средства отладки. Раскрыты механизмы управления веб-страницами: события и их обработка, управление элементами, графика и мультимедиа, веб-формы и элементы управления, регулярные выражения, навигация и управление окнами. Рассмотрена работа с HTML API и компонентное программирование: асинхронное программирование, работа с внешними данными, программная графика, объявление своих классов, создание компонентов. Освещены технологии взаимодействия с сервером: AJAX, PHP, разработка фронтендов и бэкендов, серверные сообщения.

Электронный архив на сайте издательства содержит коды всех примеров и результаты выполнения упражнений.

Для начинающих веб-разработчиков

УДК 004.438JavaScript
ББК 32.973.26-018.1

Группа подготовки издания:

Руководитель проекта	*Евгений Рыбаков*
Зав. редакцией	*Екатерина Сависте*
Компьютерная верстка	*Ольги Сергиенко*
Дизайн обложки	*Карины Соловьевой*

Подписано в печать 07.07.20.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 28,38.
Тираж 1000 экз. Заказ №11504.
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Отпечатано с готового оригинал-макета
ООО "Принт-М", 142300, М.О., г. Чехов, ул. Полиграфистов, д. 1

ISBN 978-5-9775-6589-9

© ООО "БХВ", 2021
© Оформление ООО "БХВ-Петербург", 2021

Оглавление

Введение **11**
Почему JavaScript? 11
Что вы найдете в этой книге? 12
Что вам понадобится? 12
Типографские соглашения 13

ЧАСТЬ I. НАЧАЛА JAVASCRIPT **15**

Урок 1. Типы данных, операторы и переменные **17**
1.1. Упражнение. Используем консоль веб-обозревателя 17
1.2. Типы данных и операторы 20
1.3. Числа, арифметические операторы и приоритет операторов 20
1.4. Строки. Конкатенация строк 22
1.5. Условия и операторы сравнения 23
1.6. Логические операторы 24
1.7. Переменные 24
1.7.1. Три оператора объявления переменных 27
1.7.2. Еще об объявлении переменных 27
1.8. Арифметические операторы, применимые лишь к переменным 28
1.9. Условный оператор 28
1.10. Получение типа данных 29
1.11. Преобразование типов 30
1.12. Самостоятельные упражнения 32

Урок 2. Выражения и управляющие конструкции **35**
2.1. Упражнение. Пишем первый веб-сценарий 35
2.2. Упражнение. Применяем условные выражения 38
2.2.1. Сокращенная форма условного выражения 39
2.2.2. Множественное условное выражение 39
2.3. Упражнение. Используем блоки 40
2.3.1. Переменные в блоках 41
2.4. Выражение выбора 41
2.5. Упражнение. Используем цикл со счетчиком 43
2.5.1. Объявление счетчика непосредственно в цикле 46
2.5.2. В приращении можно записать несколько выражений 46
2.6. Циклы с предусловием и постусловием 46
2.7. Операторы *break* и *continue* 47
2.8. Комментарии 48
2.9. Строгий режим 48
2.10. Как набирать JavaScript-код? 49
2.11. Самостоятельные упражнения 49

Урок 3. Функции и массивы 51
3.1. Упражнение. Используем функции 51
3.2. Локальные переменные и вложенные функции 54
3.3. Функции с необязательными параметрами 55
3.3.1. Функции с необязательными параметрами в «старом» стиле 56
3.4. Функции с произвольным количеством параметров 56
3.5. Упражнение. Используем рекурсию 57
3.6. Анонимные функции и функции-стрелки 58
3.7. Функциональный тип данных 60
3.8. Упражнение. Создаем внешний веб-сценарий 60
3.9. Встроенные функции JavaScript 61
3.10. Массивы. Объектный тип данных 61
3.11. Самостоятельные упражнения 63

Урок 4. Классы и объекты 65
4.1. Объекты. Свойства и методы 65
4.1.1. Доступ к свойствам объекта посредством оператора *[]* (квадратные скобки) 66
4.2. Классы 66
4.3. Класс *Number* 67
4.4. Класс *String* 69
4.5. Класс *Array* 71
4.5.1. Сортировка массивов 76
4.6. Класс *Date* 78
4.7. Класс *Math* 80
4.8. Класс *Arguments*. Коллекции 82
4.9. Класс *Object*. Служебные объекты. Объектная нотация 82
4.10. Объектный тип. Значение *null* 83
4.11. Хранение объектов в переменных. Значащие и ссылочные типы 84
4.12. Добавленные свойства 85
4.13. Дополнительные средства для работы с объектами 86
4.14. Самостоятельные упражнения 87

Урок 5. Средства отладки 89
5.1. Вывод сообщений в консоли 89
5.1.1. Вывод в консоли произвольных сообщений 89
5.2. Упражнение. Работаем с отладчиком веб-обозревателя 90
5.2.1. Удаление точек останова 95
5.3. Инспектор DOM 95
5.4. Исключения и их обработка 96
5.4.1. Генерирование исключений 98
5.5. Самостоятельные упражнения 99

ЧАСТЬ II. УПРАВЛЕНИЕ ВЕБ-СТРАНИЦЕЙ И ВЕБ-ОБОЗРЕВАТЕЛЕМ 101

Урок 6. События и их обработка 103
6.1. Упражнение. Обрабатываем события 103
6.1.1. Еще немного об обработчиках событий 107
6.1.2. Удаление привязки обработчика к событию 108
6.1.3. Альтернативный способ привязки обработчиков к событиям 108

6.2. События, поддерживаемые элементами страницы ... 109
6.2.1. Особенности события *beforeunload* ... 110
6.3. Упражнение. Получаем сведения о событии ... 111
6.3.1. Классы событий и их свойства ... 113
6.4. Упражнение. Изучаем фазы событий ... 116
6.4.1. Свойства и методы класса *Event*, имеющие отношение к «прохождению» событий ... 118
6.5. Упражнение. Отменяем обработку событий по умолчанию ... 118
6.5.1. Свойства класса *Event*, касающиеся обработки события по умолчанию ... 121
6.6. Самостоятельные упражнения ... 121

Урок 7. Управление элементами веб-страниц ... 123
7.1. Получение доступа к элементам страницы ... 123
7.1.1. Доступ к элементам определенного типа ... 123
7.1.2. Доступ к любому элементу страницы ... 124
7.1.3. Доступ к родителю, соседям и потомкам ... 126
7.1.4. Контекст исполнения веб-сценариев ... 127
7.2. Управление элементами веб-страницы ... 127
7.3. Упражнение. Делаем стильную полосу прокрутки ... 131
7.4. Упражнение. Управляем стилями ... 134
7.4.1. Средства для управления стилевыми классами ... 136
7.4.2. Средства для управления встроенными стилями ... 137
7.5. Изменение содержимого элементов ... 137
7.5.1. Методы *write* и *writeln* ... 139
7.6. Упражнение. Создаем новые элементы веб-страниц путем конструирования ... 139
7.6.1. Методы, конструирующие элементы веб-страниц ... 141
7.7. Получение сведений о веб-странице ... 144
7.8. Самостоятельные упражнения ... 144

Урок 8. Графика и мультимедиа ... 147
8.1. Получение сведений о графических изображениях ... 147
8.2. Управление мультимедийными элементами ... 148
8.2.1. Свойства ... 148
8.2.2. Методы ... 149
8.2.3. События ... 150
8.3. Упражнение. Реализуем свой видеопроигрыватель ... 151
8.4. Самостоятельные упражнения ... 154

Урок 9. Веб-формы и элементы управления ... 155
9.1. Взаимодействие с элементами управления ... 155
9.1.1. Получение и обработка введенного в элемент значения ... 158
9.1.2. Программное создание и удаление пунктов списка ... 159
9.2. Упражнение. Пишем первое клиентское веб-приложение ... 160
9.3. Упражнение. Работаем с веб-формой ... 162
9.3.1. Свойства, методы и события веб-формы ... 164
9.4. Упражнение. Реализуем валидацию данных в веб-форме ... 164
9.4.1. Вывод сообщений об ошибках в произвольном месте веб-страницы ... 166
9.4.2. Вывод произвольных сообщений об ошибках ... 167
9.5. Самостоятельные упражнения ... 168

Урок 10. Регулярные выражения ... 171
10.1. Введение в регулярные выражения ... 171
10.1.1. Создание регулярных выражений ... 171
10.1.2. Использование регулярных выражений: простые случаи ... 173
10.2. Литералы регулярных выражений ... 173
10.2.1. Метасимволы ... 173
10.2.2. Поднаборы ... 174
10.2.3. Вариант ... 175
10.2.4. Квантификаторы ... 175
10.2.5. Подквантификатор ... 176
10.2.6. Группы и обратные ссылки ... 177
10.2.7. Обычные символы ... 178
10.3. Поиск и обработка фрагментов, совпадающих с регулярными выражениями ... 178
10.3.1. Обычный режим поиска ... 178
10.3.2. Глобальный поиск ... 179
10.3.3. Многострочный поиск ... 180
10.3.4. Замена совпавших фрагментов ... 180
10.3.5. Прочие полезные инструменты ... 181
10.4. Упражнение. Выполняем программную валидацию с помощью регулярного выражения ... 181
10.5. HTML-валидация с применением регулярных выражений ... 182
10.6. Самостоятельные упражнения ... 182

Урок 11. Взаимодействие с веб-обозревателем ... 185
11.1. Работа с окном веб-обозревателя ... 185
11.1.1. Еще один способ записи веб-сценариев, манипулирующих элементами страницы ... 187
11.2. Упражнение. Навигация по якорям с подсветкой активной гиперссылки ... 188
11.3. Работа с текущим интернет-адресом ... 190
11.4. Получение сведений о клиентском компьютере ... 191
11.5. Вывод стандартных диалоговых окон ... 192
11.6. Самостоятельное упражнение ... 193

ЧАСТЬ III. HTML API И КОМПОНЕНТНОЕ ПРОГРАММИРОВАНИЕ ... 195

Урок 12. Таймеры и фоновые потоки ... 197
12.1. Упражнение. Используем периодические таймеры ... 197
12.2. Однократный таймер ... 201
12.3. Упражнение. Применяем фоновые потоки ... 202
12.4. Самостоятельные упражнения ... 205

Урок 13. Работа с файлами, хранение данных и перетаскивание ... 207
13.1. Работа с локальными файлами ... 207
13.1.1. Чтение сведений о файлах ... 207
13.1.2. Считывание текстовых файлов. Класс *FileReader* ... 208
13.1.3. Вывод индикатора процесса при считывании файла ... 210
13.1.4. Считывание графических файлов ... 210
13.2. Упражнение. Реализуем предварительный просмотр выбранного графического файла ... 210
13.3. Хранение данных на стороне клиента ... 212

13.4. Перетаскивание ... 213
13.4.1. Превращение элемента-источника в перетаскиваемый ... 213
13.4.2. Задание перемещаемых данных в источнике ... 213
13.4.3. Указание допустимых операций ... 214
13.4.4. Подготовка элемента-приемника ... 215
13.4.5. Завершение перетаскивания ... 216
13.5. Упражнение. Практикуемся в реализации перетаскивания ... 216
13.6. Перетаскивание файлов в поле выбора файлов ... 219
13.7. Самостоятельные упражнения ... 219

Урок 14. Программная графика ... 221
14.1. Холст HTML ... 221
14.2. Рисование прямоугольников ... 222
14.3. Указание основных параметров графики ... 222
14.3.1. Цвета линий и заливок ... 223
14.3.2. Параметры тени ... 223
14.4. Вывод текста ... 224
14.5. Упражнение. Рисуем диаграмму ... 225
14.6. Рисование сложных фигур ... 227
14.6.1. Начало и завершение рисования ... 227
14.6.2. Перемещение пера ... 227
Рисование прямых линий ... 228
Замыкание контура ... 228
Указание параметров линий ... 229
Рисование дуг ... 230
Рисование кривых Безье ... 231
Рисование прямоугольников ... 232
14.7. Градиенты и графические закраски ... 233
14.7.1. Линейные градиенты ... 233
14.7.2. Радиальные градиенты ... 235
14.7.3. Графическая закраска ... 236
14.8. Преобразования ... 237
14.9. Вывод сторонних изображений ... 238
14.10. Управление композицией ... 239
14.11. Создание масок ... 240
14.12. Сохранение и восстановление состояния холста ... 240
14.13. Упражнение. Рисуем цветок ... 241
14.14. Самостоятельные упражнения ... 242

Урок 15. Объявление своих классов ... 245
15.1. Объявление нового класса ... 245
15.1.1. Конструктор. Объявление свойств и методов в конструкторе ... 245
15.1.2. Прототип. Объявление свойств и методов в прототипе ... 246
15.2. Упражнение. Объявляем класс слайдера ... 247
15.3. Упражнение. Создаем динамическое свойство ... 251
15.3.1. Динамические свойства, доступные только для чтения и только для записи ... 252
15.3.2. Объявление обычных свойств методом *defineProperty* ... 252
15.4. Упражнение. Реализуем наследование классов ... 253
15.4.1. Переопределение методов ... 255
15.4.2. Использование метода *call* для вызова методов базового класса ... 256

15.5. Объявление статических свойств и методов 256
15.6. Упражнение. Расширяем функциональность встроенных классов 257
15.7. Самостоятельные упражнения 258

Урок 16. Компоненты 261
16.1. Понятие компонента 261
16.2. Упражнение. Создаем компонент 261
16.3. Замыкания 265
16.4. Упражнение. Изолируем компонент в замыкании 267
16.5. Самостоятельные упражнения 268

ЧАСТЬ IV. ВЗАИМОДЕЙСТВИЕ С СЕРВЕРОМ 269

Урок 17. Технология AJAX 271
17.1. Реализация AJAX 271
17.1.1. Подготовка объекта AJAX 272
17.1.2. Указание интернет-адреса загружаемого файла 272
17.1.3. Получение загруженного файла 272
17.1.4. Отправка веб-серверу запроса на получение файла 273
17.2. Упражнение. Пишем сверхдинамический веб-сайт 274
17.3. Синхронная загрузка файлов по технологии AJAX 276
17.4. Формат JSON 276
17.5. Упражнение. Загрузка и вывод JSON-данных 277
17.6. Упражнение. Пишем компонент *AJAXLoader* 280
17.7. Самостоятельные упражнения 282

Урок 18. Серверные веб-приложения. Платформа PHP 285
18.1. Серверные веб-приложения. Платформа PHP 285
18.2. Упражнение. Изучаем основы PHP 286
18.3. Упражнение. Пишем простейшую фотогалерею на PHP 289
18.4. Упражнение. Передаем данные методом GET 293
18.5. Упражнение. Передаем данные методом POST 296
18.6. Самостоятельные упражнения 300

Урок 19. Разработка фронтендов и бэкендов 301
19.1. Веб-разработка: старый и новый подходы. Фронтенды и бэкенды. Веб-службы 301
19.2. Упражнение. Новая фотогалерея. Вывод списка миниатюр 302
19.3. Упражнение. Новая фотогалерея. Показ изображений 305
19.4. Отправка веб-службе данных методом POST 309
19.4.1. Отправка веб-формы целиком 309
19.4.2. Отправка произвольных данных 310
19.4.3. Отправка данных в виде строки POST-параметров 310
19.5. Упражнение. Новая фотогалерея. Загрузка изображений в галерею 311
19.6. Самостоятельные упражнения 316

Урок 20. Серверные сообщения 317
20.1. Использование серверных сообщений 317
20.1.1. Что представляет собой серверное сообщение? 317
20.1.2. Отправка серверных сообщений 318
20.1.3. Прием серверных сообщений 318

20.1.4. Поддержание соединения и возобновление приема ... 319
20.1.5. Создание своих событий ... 320
20.1.6. Разрыв соединения ... 320
20.2. Упражнение. Новая фотогалерея. Более отзывчивый список миниатюр ... 321

Заключение ... 325

Приложение 1. Приоритет операторов ... 327

Приложение 2. Пакет хостинга XAMPP ... 329
П2.1. Установка пакета хостинга XAMPP ... 329
П2.2. Панель управления XAMPP. Запуск и остановка веб-сервера ... 334
П2.2.1. Указание языка при первом запуске ... 334
П2.2.2. Окно панели управления XAMPP ... 334
П2.2.3. Запуск веб-сервера ... 335
П2.2.4. Проверка работоспособности веб-сервера ... 335
П2.2.5. Остановка веб-сервера ... 336
П2.3. Использование веб-сервера ... 336
П2.3.1. Тестирование веб-сайта с применением XAMPP ... 336
П2.3.2. Просмотр журналов работы веб-сервера и PHP ... 336
П2.4. Решение проблем ... 337
П2.4.1. Увеличение максимального размера выгружаемых файлов ... 337
П2.4.2. Отключение кеширования файлов веб-обозревателем ... 338

Приложение 3. Описание электронного архива ... 341

Предметный указатель ... 343

Введение

Почему JavaScript?
Что вы найдете в этой книге?
Что вам понадобится в процессе чтения?
Типографские соглашения

Здравствуйте, уважаемый читатель!

Если вы держите в руках эту книгу, значит, желаете научиться языку программирования JavaScript. Это очень правильное желание. А это очень правильная книга!

JavaScript — один из популярнейших в настоящее время языков для написания программ, а в деле веб-программирования у него вообще нет конкурентов. Знание JavaScript требуется от каждого, кто намеревается серьезно заняться написанием сайтов. Любой работодатель, узнавший, что претендент на должность веб-разработчика не владеет этим языком, тут же даст ему от ворот поворот.

Почему JavaScript?

Прежде всего, он — один из трех «китов», на которых зиждется величественный дворец веб-разработки. Первый «кит» — язык HTML — описывает саму веб-страницу, ее содержание, второй — CSS — ее оформление, а JavaScript — поведение.

JavaScript может «оживить» страницу, заставляя ее реагировать на действия пользователя. Он может превратить набор обычных картинок в красивую фотогалерею. Он может загрузить с сервера какие-либо данные и вывести их прямо на странице в виде абзаца, списка, таблицы или графика. Он даже может превратить страницу в настоящую программу, обрабатывающую занесенные пользователем данные и выводящую результаты их обработки.

JavaScript — единственный язык программирования, позволяющий «влезть» внутрь веб-страницы, исправить или дополнить ее содержание. JavaScript — единственный язык программирования, непосредственно поддерживаемый веб-обозревателями.

Ранее говорилось, что у этого языка в своей области нет конкурентов. И в ближайшем будущем они не появятся.

Что вы найдете в этой книге?

Философия программирования — это часть программирования, со всех сторон окруженная водой.

- Двадцать коротких, емких, наглядных, иллюстрированных уроков, дающих необходимые теоретические знания.
- Более сорока практических упражнений, выполняемых под руководством автора. Выполняя их, вы закрепите полученные знания и приобретете программистский опыт, что безусловно оценит ваш будущий работодатель.
- Восемнадцать упражнений, рассчитанных на самостоятельное выполнение. Чтобы справиться с ними, необходимо лишь внимательно изучать приведенный в уроках теоретический материал.
- Описание самых востребованных в настоящее время приемов программирования: объявления своих классов, написания компонентов, программирования фронтендов и бэкендов, использования фоновых процессов и серверных сообщений.
- Вводный курс популярной программной платформы PHP, знание которой очень пригодится веб-программисту.
- Полное отсутствие описаний устаревших и малополезных инструментов, отвлеченных рассуждений, философствований на тему программирования и прочей «воды».

> *Чтобы по этой книге овладеть JavaScript-программированием, вам хватит и месяца.*

Книгу сопровождает электронный архив (см. *приложение 3*), содержащий результаты выполнения всех упражнений, равно как и необходимые исходные файлы. Архив выложен на FTP-сервер издательства «БХВ-Петербург» по адресу **ftp://ftp.bhv.ru/9785977565899.zip**. Ссылка доступна и со страницы книги на сайте издательства **www.bhv.ru**.

Что вам понадобится?

Для изучения этой книги вам, уважаемый читатель, нужны следующие программы:

- *веб-обозреватель*, разумеется. Автор тестировал подготовленные им примеры в Google Chrome и Mozilla Firefox (остальные веб-обозреватели либо аналогичны Chrome, либо малопопулярны);
- *текстовый редактор* — можно использовать Блокнот, но автор рекомендует какой-либо из чисто «программистских» редакторов: Visual Studio Code (**https://code.visualstudio.com/**), Atom (**https://atom.io/**), Notepad++ (**https://notepad-plus-plus.org/**), Sublime Text (**https://www.sublimetext.com/**), Brackets (**http://brackets.io/**) и т. п.;

- *пакет веб-хостинга XAMPP* (**https://www.apachefriends.org/ru/index.html**). Автор применял версию 7.3.5, включающую веб-сервер Apache HTTP Server 2.4.39 и PHP 7.3.5. Описание пакета веб-хостинга XAMPP и инструкция по его установке на компьютер приведены в *приложении 2*.

Еще от вас требуется:

- владеть языками HTML и CSS, на которых пишутся сами веб-страницы и оформление для них;
- иметь минимальные навыки вёб-верстки;
- знать в общих чертах как работает веб-сервер (это понадобится нам в *части IV* книги).

Типографские соглашения

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

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

```
<script type="text/javascript">
    const ins = window.prompt('Величина в дюймах', 0);
    const cents = ins * 2.54;
</script>
```

- в угловые скобки <> заключаются наименования различных значений, которые дополнительно выделяются *курсивом*. В реальный код, разумеется, должны быть подставлены реальные значения. Например:

```
return <возвращаемое значение>
```

  Здесь вместо подстроки *возвращаемое значение* должно быть подставлено реальное возвращаемое значение;

- в квадратные скобки [] заключаются необязательные фрагменты кода. Например:

```
toString([<основание>])
```

  Здесь *основание* может указываться, а может и не указываться;

- слишком длинные, не помещающиеся на одной строке книги фрагменты, автор разрывал на несколько строк и в местах разрывов ставил знаки ↳. Например:

```
active = document.querySelector↳
('nav a.active');
```

  Приведенный код здесь разбит на две строки, но должен быть набран в одну. Символ ↳ при этом нужно удалить;

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

```
function circleLength(d) {
    d = d || 2;
    . . .
}
```

Здесь весь код между второй и последней строками пропущен.

Обычно такое можно встретить в исправленных впоследствии фрагментах кода — приведены лишь собственно исправленные строки, а оставшиеся неизмененными пропущены. Также троеточие используется, чтобы показать, в какое место должен быть вставлен вновь написанный код: в начало исходного фрагмента, в его конец или в середину, между уже присутствующими в нем строками.

Внимание!

Все приведенные здесь типографские соглашения имеют смысл только в примерах написания языковых конструкций JavaScript.

ЧАСТЬ I

Начала JavaScript

- Данные, которыми манипулирует JavaScript.
- Переменные.
- Операторы.
- Выражения.
- Управляющие конструкции.
- Функции.
- Объекты и классы.
- Веб-сценарии: внутренние и внешние.
- Средства отладки.
- Исключения и обработка ошибок.

Урок 1
Типы данных, операторы и переменные

Консоль веб-обозревателя
Типы данных
Числа
Строки
Логические значения
Операторы
Переменные
Преобразование типов

1.1. Упражнение. Используем консоль веб-обозревателя

Изучение основ языка JavaScript удобно начать, набирая код в консоли.

> *Консоль* — интерактивная исполняющая среда, встроенная в веб-обозреватель. В консоли можно набрать какое-либо выражение и сразу увидеть результат его выполнения.

Вызовем консоль у установленного у нас веб-обозревателя и выполним в ней несколько выражений.

1. Запустим веб-обозреватель (автор книги использовал Google Chrome).
2. Нажмем клавишу <F12>, чтобы вызвать встроенные в веб-обозреватель отладочные инструменты, в число которых входит и консоль. По умолчанию панель с отладочными инструментами появляется у нижней стороны окна веб-обозревателя (рис. 1.1).
3. В верхней части панели находятся корешки вкладок, на которых располагаются различные отладочные инструменты. Переключимся на вкладку **Console**, щелкнув на ней мышью (рис. 1.2).

   На этой вкладке и находится консоль веб-обозревателя (рис. 1.3).

   В белой области редактирования, занимающей бóльшую часть вкладки, виден мигающий текстовый курсор. Здесь вводятся выражения JavaScript, которые нужно выполнить.

- Для начала сложим два числа, для чего наберем в области редактирования следующий простой код:

```
27 + 14
```

и нажмем клавишу <Enter>.

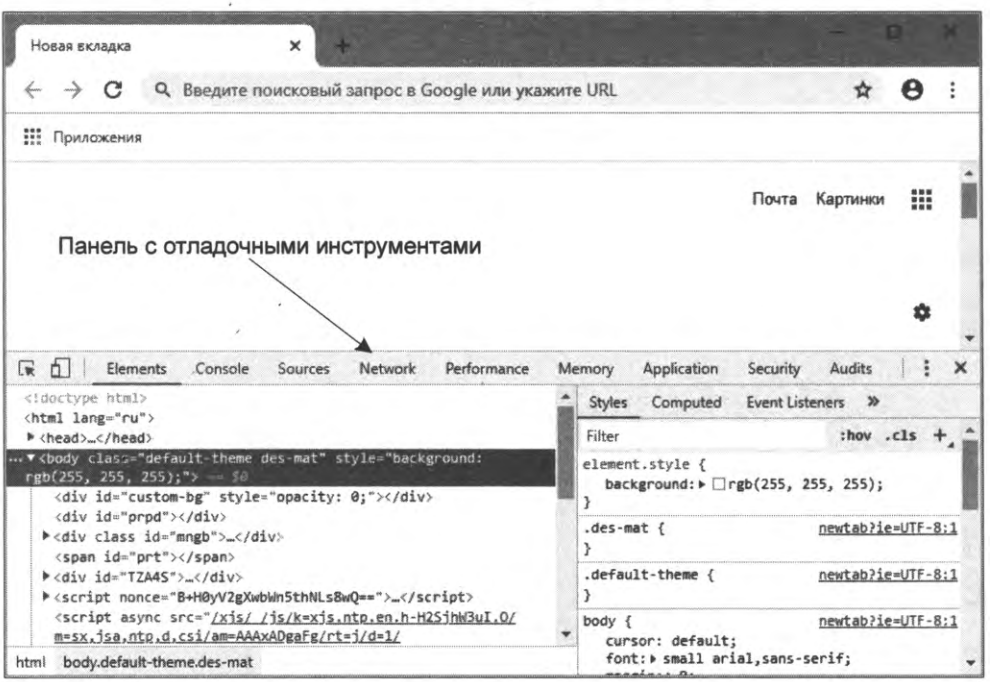


Рис. 1.1. Панель с отладочными инструментами веб-обозревателя

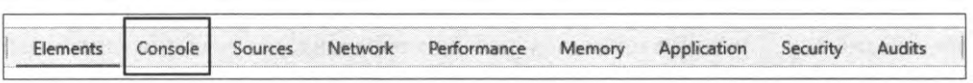


Рис. 1.2. Корешок вкладки **Console** на панели с отладочными инструментами веб-обозревател

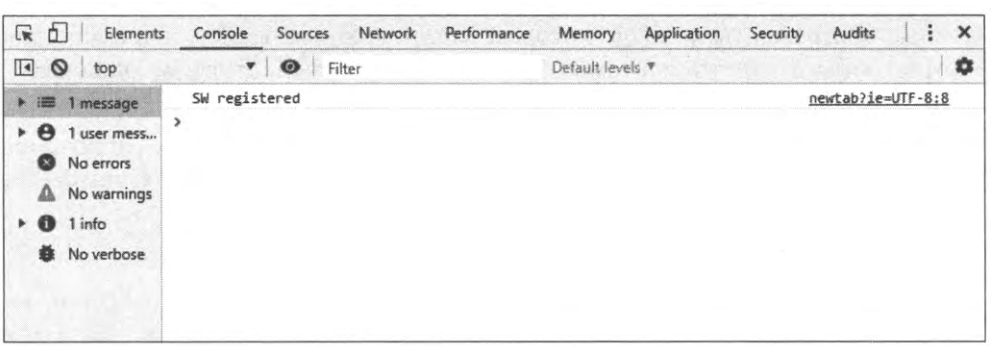


Рис. 1.3. Консоль веб-обозревателя

Ниже набранного кода мы увидим результат сложения чисел 27 и 14 (рис. 1.4).

Символом «больше» >, расположенным в консоли слева, помечается строка, в которую в настоящий момент будет вводиться код JavaScript.

```
SW registered
> 27 + 14
< 41
>
```

Рис. 1.4. Результат сложения чисел 27 и 14 в консоли веб-обозревателя

Символом «меньше» < помечается строка, в которой выводится результат исполнения набранного кода после нажатия клавиши <Enter>.

- Мы можем обрабатывать и строки: давайте объединим строки `Java` и `Script` в одну:

```
> 'Java' + 'Script'
< "JavaScript"
```

Здесь символ + обозначает операцию объединения строк.

- Еще мы можем сравнивать значения: проверим, действительно ли число 10 меньше 20:

```
> 10 < 20
< true
```

Результатом будет логическое значение `true` — «истина». Так веб-обозреватель сообщает нам, что записанное сравнение выполняется (оно истинно).

- Напоследок для эксперимента выполним заведомо не выполняющееся, ложное сравнение:

```
> 15 > 25
< false
```

Результатом станет логическое значение `false` — «ложь».

4. Очистим консоль, щелкнув на кнопке **Clear console**, что находится в окне консоли слева, под набором корешков вкладок (рис. 1.5). Для очистки консоли можно также нажать комбинацию клавиш <Ctrl>+<L>.

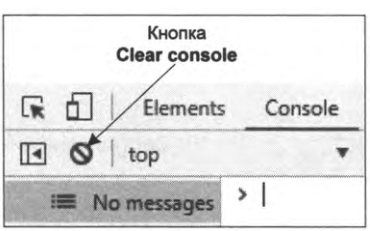


Рис. 1.5. Кнопка очистки консоли **Clear console**

1.2. Типы данных и операторы

Любое значение, обрабатываемое JavaScript, относится к определенному типу данных.

> *Тип данных* — определяет характер значения и набор выполняемых над ним операций.

В *разд. 1.1* мы познакомились с тремя типами данных (табл. 1.1).

Таблица 1.1. *Три типа данных JavaScript*

Числовой	Строковый	Логический
Обычные числа	Строки из любых символов	Логические значения
`14, 27, 10000`	`'Java', 'Script', 'HTML и CSS'`	`true` — «истина» `false` — «ложь»

Там же, в *разд. 1.1*, для обработки значений мы использовали операторы.

> *Оператор* — выполняет элементарное действие над заданными значениями (*операндами*): сложение, объединение строк, сравнение и пр. — и производит выдачу (*возврат*) результата.

Примеры:

```
> 27 + 14
< 41
```
Оператор сложения + складывает операнды `27` и `14`
После чего возвращает результат — сумму `41`

```
> 10 < 20
< true
```
Оператор «меньше» < сравнивает операнды `10` и `20`
И возвращает результат — логическое значение `true`

Некоторые операторы JavaScript не принимают операндов и (или) не возвращают результат.

1.3. Числа, арифметические операторы и приоритет операторов

Числа в языке JavaScript записываются по обычным (школьным) правилам. Но в дробных числах (их еще называют *вещественными* и *числами с плавающей точкой*) в качестве десятичного разделителя применяется точка (не запятая!).

Примеры:

Целые числа: `10, 20, 1234567` Вещественные числа: `1.5, 23.67, 0.053`

У отрицательных чисел в начале ставится знак «минус»: -. Для записи чисел в экспоненциальной форме: *<мантисса>* × $10^{<порядок>}$ — применяется формат:

*<мантисса>*e*<порядок>*

Примеры:

Отрицательные числа:	Числа в экспоненциальной форме:
`-10`, `-20`, `-0.32`, `-0.053`	`1e6` (1000000), `1.3e-5` (0,000013), `-1e3` (–1000)

Помимо десятичных, можно записывать восьмеричные и шестнадцатеричные числа, предварив их, соответственно, нулем или символами `0x`.

Примеры:

Восьмеричные числа:	Шестнадцатеричные числа:
`010` (8), `0123` (83), `-045` (–37)	`0x10` (16), `0x53` (83), `-0x321` (–801)

Для выполнения арифметических действий применяются *арифметические операторы*.

> *Бинарные* (принимающие два операнда) операторы записываются в формате:
> *<операнд 1> <оператор> <операнд 2>*

> ***Пояснение***
> Операторы бывают *унарные* (принимающие один операнд), *бинарные* (принимающие два операнда) и *тернарные* (принимающие три операнда). Обо всех их подробно рассказывается по мере их упоминания.

Поддерживаемые JavaScript арифметические операторы приведены в табл. 1.2.

Таблица 1.2. *Поддерживаемые JavaScript арифметические операторы*

Оператор	Описание	Пример	Результат
+ (плюс)	Сложение	`> 27 + 14`	`< 41`
- (минус)	Вычитание	`> 40 - 100`	`< -60`
* (звездочка)	Умножение	`> 12 * 32`	`< 384`
/ (слеш)	Деление	`> 3 / 4`	`< 0.75`
% (процент)	Остаток от деления	`> 4 % 3`	`< 1`
**	Возведение в степень	`> 2 ** 10`	`< 1024`

> Очередность выполнения операторов задается их *приоритетом*: операторы с бóльшим приоритетом выполняются раньше операторов с меньшим приоритетом.

Из рассмотренных ранее операторов наибольший приоритет имеет возведение в степень: `**`, операторы умножения: `*`, деления: `/` и остатка от деления: `%` — меньший, а сложения: `+` и вычитания: `-` — еще меньший (таблицу с приоритетами разных операторов можно найти в *приложении 1*).

- В следующем примере сначала будет выполнено деление `82` на `2`, а потом полученное частное сложится с `37`:

```
> 37 + 82 / 2
< 78
```

- Операторы с одинаковым приоритетом выполняются в порядке слева направо, поэтому в следующем примере сначала будет выполнено сложение `10` и `34`, а потом из суммы вычтется `26`:

```
> 10 + 34 - 26,
< 18
```

- Порядок выполнения операторов можно менять с помощью обычных круглых скобок. Оператор в круглых скобках будет выполнен раньше, независимо от его приоритета. Поэтому здесь сначала будет выполнено сложение `37` и `82`, а потом полученная сумма разделится на `2`:

```
> (37 + 82) / 2
< 59.5
```

К числам также относятся три специальных значения:

- `NaN` (Not a Number, не число) — значение, получаемое при попытке, например, умножить число на строку (о строках будет рассказано позже):

```
> 20 * 'JavaScript'
< NaN
```

- `Infinity` — математическая «бесконечность»: ∞. Это значение можно получить делением положительного числа на `0`:

```
> 10 / 0
< Infinity
```

- `-Infinity` — математическая «минус бесконечность»: –∞. Может быть получена делением отрицательного числа на `0`.

1.4. Строки. Конкатенация строк

> *Строки* заключаются в одинарные или двойные прямые кавычки (автор книги предпочитает одинарные).

Примеры строк: `'Java'`, `"Script"`, `'Языки HTML и CSS применяются в веб-разработке'`

Если строка заключена в одинарные кавычки, в ней не допускаются символы одинарной кавычки. Если строка заключена в двойные кавычки, в ней не допускаются символы двойной кавычки. В любом случае в строках не допускаются символы обратного слеша.

Примеры:

Правильно:	Неправильно:
`"Шеймус O'Брайен"`	`'Шеймус O'Брайен'`
`'Суши-бар "Йокогама"'`	`"Суши-бар "Йокогама""`

Указанные запреты можно обойти, применив *строковые литералы* (специальные последовательности символов): `\'`, `\"` — для кавычек и `\\` — для обратного слеша.

Примеры использования литералов:

```
'Шеймус О\'Брайен'
"Суши-бар \"Йокогама""
```

Последовательность литералов `\r\n` вставляет в строковое значение разрыв строки.

Оператор +, будучи примененным к операндам-строкам, выполняет объединение (*конкатенацию*) строк в одну.

Объединяем строки `'Java'` и `'Script'`:

```
> 'Java' + 'Script'
< "JavaScript"
```

1.5. Условия и операторы сравнения

Для того чтобы сравнить какие-либо значения, применяется условие.

> *Условие* — языковая конструкция, сравнивающая заданные значения. Записывается с помощью оператора сравнения.

> *Оператор сравнения* — сравнивает два операнда и возвращает результат в виде логического значения `true` («истина») или `false` («ложь»).

Операторы сравнения являются *бинарными* и записываются в формате:
<операнд 1> <оператор> <операнд 2>

Все поддерживаемые JavaScript операторы сравнения приведены в табл. 1.3.

***Таблица 1.3.** Поддерживаемые JavaScript операторы сравнения*

Оператор	Описание	Пример	Результат
==	Равно[1]	> 1 == 1.000	< true
!=	Не равно	> 1 != 1.001	< true
<	Меньше	> 1000 < 100	< false
<=	Меньше или равно	> 100 <= 1000	< true
>	Больше	> 10 > 9	< true
>=	Больше или равно	> 1e6 >= 1e9	< false
===	Строго равно		
!==	Строго не равно		

Два последних оператора мы рассмотрим в конце урока, когда будем разбираться с преобразованием типов.

[1] Два знака «равно». Один знак «равно» — это совсем другой оператор, который будет рассмотрен позже.

1.6. Логические операторы

Логический оператор — принимает логические операнды и возвращает результат также в виде логического значения.

В JavaScript имеются три логических оператора:

- `!` (*логическое отрицание*, или *логическое НЕ*) — *унарный*, возвращает `false`, если *операнд* имеет значение `true`, и `true` в противном случае. Записывается этот оператор в формате: `!<операнд>` без пробела. Пример:

```
> !(10 == 100)
< true
```

  В выражениях с оператором `!` операторы сравнения всегда следует брать в круглые скобки. Приоритет оператора `!` выше, чем у операторов сравнения, следовательно, он будет выполнен первым, и результат без скобок окажется неверным;

- `&&` (*логическое умножение*, или *логическое И*) — возвращает `true`, если оба *операнда* равны `true`, и `false` в противном случае;

- `||` (*логическое сложение*, или *логическое ИЛИ*) — возвращает `true`, если хотя бы один *операнд* равен `true`, и `false` в противном случае.

  Два последних оператора записываются в формате:

```
<операнд 1> <оператор && или ||> <операнд 2>
```

  Примеры:

```
> 10 <= 100 && 0 != 1
< true

> 10 >= 100 && 0 != 1
< false

> 10 <= 100 && 0 == 1
< false

> 10 >= 100 && 0 == 1
< false
```

```
> 10 <= 100 || 0 != 1
< true

> 10 >= 100 || 0 != 1
< true

> 10 <= 100 || 0 == 1
< true

> 10 >= 100 || 0 == 1
< false
```

1.7. Переменные

Переменная — ячейка памяти, предназначенная для временного хранения какого-либо значения. Должна иметь уникальное имя, по которому выполняется обращение к этой переменной.

Имя переменной может содержать буквы латиницы, цифры и символы подчеркивания _, причем оно **не** должно начинаться с цифры. Пробелы в имени переменной не допускаются.

Примеры:

Правильные имена:	Неправильные имена:
a, b, someValue, _private	число, some value, 234value

Перед первым использованием переменной ее следует объявить.

> *Объявление переменной* — указание создать переменную с заданным именем. Должно предшествовать первому использованию переменной.

Сразу при объявлении переменной она получает значение `undefined`, обозначающее отсутствие какой-либо значащей величины.

Оператор объявления переменных `let` объявляет переменные с указанными *именами*:

```
let <список имен объявляемых переменных через запятую>
```

В качестве результата оператор `let` всегда возвращает значение `undefined`.

Для объявления переменных можно использовать еще два оператора, которые мы рассмотрим в конце этого раздела.

Объявляем переменную с именем `sum` (и получим результат `undefined`):

```
> let sum
< undefined
```

> ***Примечание***
>
> В дальнейшем результат `undefined`, возвращаемый оператором `let`, ради компактности указываться не будет.

Объявляем сразу три переменные:

```
> let x, y, x
```

Объявив переменную, мы можем присвоить ей сохраняемое значение.

> *Присваивание* — занесение значения в переменную. Выполняется *оператором присваивания* =[1], имеющим формат:
> *<переменная> = <значение>*

Присваиваем только что объявленной переменной `sum` значение `10`:

```
> sum = 10
< 10
```

В качестве результата оператор = возвращает присвоенное переменной значение.

Можно совместить объявление переменной с присваиванием ей значения — так мы сократим код. Для этого в операторе `let` вместо имени переменной нужно указать оператор присваивания.

Объявляем переменную `a` и сразу же присваиваем ей число `1`:

```
> let a = 1
```

Сохраненное в переменной значение можно использовать в вычислениях, выполнив обращение к этой переменной.

---

[1] Оператор присваивания обозначается одним знаком «равно» (=), а оператор сравнения «равно» — двумя (==). Не путайте эти два оператора!

Обращение к переменной — указание извлечь хранящееся в переменной значение и использовать его в качестве операнда какого-либо оператора или иным образом (например, для вывода на экран).

Обращение к переменной выполняется простым указанием имени нужной переменной в качестве операнда.

Присваиваем переменной `a` произведение значения переменной `sum` (в текущий момент: `10`) и числа `20`:

```
> a = sum * 20
< 200
```

Внимание!

При присваивании переменной нового значения старое значение теряется.

Прибавляем к значению переменной `sum` (в текущий момент: `10`) число `20` и заносим получившуюся сумму в ту же переменную:

```
> sum = sum + 20
< 30
```

Оператор комбинированного присваивания (табл. 1.4) извлекает из указанной *переменной* значение, выполняет над ним и заданным *операндом* какое-либо действие и присваивает результат той же *переменной*. Формат этого оператора:

<переменная> <оператор> <операнд>

Таблица 1.4. *Операторы комбинированного присваивания*

Оператор комбинированного присваивания	Его аналог
`a += b`	`a = a + b`
`a -= b`	`a = a - b`
`a *= b`	`a = a * b`
`a /= b`	`a = a / b`
`a %= b`	`a = a % b`

Прибавим к переменной `sum` (сейчас там `30`) число `70`:

```
> sum += 70
< 100
```

Операторы комбинированного присваивания выполняются быстрее аналогичных комбинаций арифметического оператора и обычного оператора присваивания.

1.7.1. Три оператора объявления переменных

Язык JavaScript поддерживает три *оператора объявления переменных*: уже знакомый нам `let`, а также `const` и `var`. Они имеют один и тот же формат записи, но разные особенности:

- `let` — объявляет обычную переменную, которой можно присваивать значения сколько угодно раз. Повторное объявление переменной этим оператором не допускается — оно вызовет ошибку:

  ```
  > let x
  > let x
  < Uncaught SyntaxError: Identifier 'x' has already
      been declared at <anonymous>:1:1
  ```

  Здесь в последних двух строках выведено сообщение об ошибке;

- `const` — объявляет *константу* — переменную, значение которой можно присвоить лишь один раз и лишь непосредственно при ее объявлении. Попытка присвоить константе новое значение вызовет ошибку. Пример:

  ```
  > const y = 2
  ```

  Если значение переменной не меняется в дальнейшем, рекомендуется объявлять ее как константу, поскольку константы обрабатываются веб-обозревателем быстрее;

- `var` — полностью аналогичен оператору `let`, однако позволяет объявить переменную повторно (в этом случае он ничего не делает). Есть еще один нюанс, который мы рассмотрим в *уроке 2*.

  Оператор `var` остался от старых версий JavaScript. В настоящее время его применение не рекомендовано, однако он часто встречается в старом JavaScript-коде.

1.7.2. Еще об объявлении переменных

Объявлять переменные перед их использованием необязательно. Можно просто присвоить переменной какое-либо значение — и веб-обозреватель сам неявно объявит эту переменную:

```
sum = 10
```

Однако все же лучше объявлять переменные явно. Во-первых, это хороший стиль программирования, а во-вторых, при работе в строгом режиме (о котором будет рассказано в *уроке 2*) веб-обозреватель будет требовать явного объявления переменных.

1.8. Арифметические операторы, применимые лишь к переменным

JavaScript поддерживает три арифметических оператора, которые принимают в качестве операндов только переменные (применение их к обычным числам вызовет ошибку). Это *унарные* (принимающие один операнд) операторы, которые записываются в формате: *<оператор><операнд>*:

- `-` (минус) — *унарный минус*, изменяет знак числа и возвращает его в качестве результата:

```
> let n = 100
> -n
< -100
```

- `++` (два плюса) — оператор *инкремента* — увеличения значения переменной на `1`. Инкрементированное число вновь заносится в ту же переменную и одновременно возвращается в качестве результата:

```
> let i = 1
> ++i
< 2

> let j = ++i
> j
< 3
```

- `--` (два минуса) — оператор *декремента* — уменьшения значения переменной на `1`. В остальном аналогичен оператору инкремента.

Операторы инкремента и декремента можно ставить и после переменной. В этом случае они сначала вернут в качестве результата текущее значение переменной, а уже потом выполнят инкремент или декремент:

```
> i++
< 3

> i
< 4
```

1.9. Условный оператор

В *разд. 1.5* мы познакомились с условиями, но лишь теоретически. Применить их на практике можно, например, в условном операторе.

> *Условный оператор* — возвращает *результат true*, если указанное *условие* истинно, и *результат false* — если оно ложно. Формат оператора:
> *(<условие>) ? <результат true> : <результат false>*

Условный оператор — единственный *тернарный* (принимающий три операнда) оператор в JavaScript.

Если строка в переменной `s` пуста, заносим в переменную `result` строку `'Строка пуста'`, в противном случае — строку `'Строка не пуста'`:

```
> let s = ''
> let result
> result = (s == '') ? 'Строка пуста' : 'Строка не пуста'
> result
< "Строка пуста"
```

А теперь попробуем с непустой строкой:

```
> s = 'Условный оператор'
> result = (s == '') ? 'Строка пуста' : 'Строка не пуста'
> result
< "Строка не пуста"
```

Вместо параметра *условие* в условном операторе можно указать переменную, хранящую логическую величину.

1.10. Получение типа данных

> *Оператор получения типа* `typeof` позволяет выяснить, к какому типу относится *значение*. Формат оператора:
>
> `typeof <значение>`

В качестве результата возвращается одна из приведенных далее строк:

- `'number'` — числовой тип:

  ```
  > typeof 123
  < "number"
  ```

- `'string'` — строковый тип:

  ```
  > let str = 'Строка'
  > typeof str
  < "string"
  ```

- `'boolean'` — логический тип:

  ```
  > typeof (24 > 90)
  < "boolean"
  ```

- `'undefined'` — *неопределенный тип*, поддерживающий всего одно значение: `undefined`.

  Оно хранится в переменной, которая была объявлена, но еще не получила какого-либо «значащего» значения:

  ```
  > let nnn
  > typeof nnn
  < "undefined"
  ```

К неопределенному типу относится и любая еще не объявленная переменная, если к ней обратиться в консоли:

```
> typeof nnnn
< "undefined"
```

> ***Внимание!***
>
> При обращении к необъявленной переменной в консоли возвращается значение `undefined`. Но если это сделать в веб-сценарии (написанием которых мы займемся на *уроке 2*), возникнет ошибка, и выполнение сценария будет прервано.

- `'function'` — функциональный тип (будет рассмотрен в *уроке 3*);
- `'object'` — объектный тип (будет рассмотрен в *уроке 4*).

1.11. Преобразование типов

Ранее мы выполняли арифметические операции над двумя числами и конкатенацию двух строк. Но что случится при попытке, например, сложить число и строку?

Тогда JavaScript предварительно произведет *преобразование типов*, чтобы привести операнды к единому типу.

Преобразование выполняется согласно следующим правилам:

- при сложении числа и строки — преобразует число в строку и производит их конкатенацию:

```
> 20 + '20'
< "2020"
```

- при выполнении любого другого арифметического действия над числом и строкой — пытается преобразовать строку в число. При этом:
  - строки, содержащие только цифры и десятичную точку, будут успешно преобразованы в числа:

```
> 20 - '10'
< 10

> 50 * '1.5'
< 75
```

  - все прочие строки будут преобразованы в значение `NaN`, и результатом вычисления также будет `NaN`:

```
> 20 - 'abc'
< NaN
```

- при выполнении арифметического действия над числом и логической величиной — значение `true` преобразуется в `1`, а `false` — в `0`:

```
> 20 + true
< 21
```

- при выполнении арифметического действия над числом и значением `undefined` — последнее преобразуется в `NaN`;
- при конкатенации строки и числового значения `NaN`, `Infinity` или `-Infinity` — последнее преобразуется в строку `'NaN'`, `'Infinity'` или `'-Infinity'` соответственно;
- при конкатенации строки и логической величины — значение `true` будет преобразовано в строку `'true'`, а `false` — в `'false'`:

```
> 'Значение: ' + false
< "Значение: false"
```

- при конкатенации строки и `undefined` — последнее преобразуется в строку `'undefined'`;
- при использовании в качестве условия значения, не относящегося к логическому типу:
  - любое ненулевое число преобразуется в `true`:

```
> (123) ? 'Ненулевое число' : 'Ноль'
< "Ненулевое число"
```

  - `0` преобразуется в `false`;
  - `Infinity` и `-Infinity` — в `true`;
  - `NaN` — в `false`;
  - любая непустая строка — в `true`:

```
> ('0') ? 'Непустая строка' : 'Пустая строка'
< "Непустая строка"
```

  - пустая строка — в `false`:

```
> ('') ? 'Непустая строка' : 'Пустая строка'
< "Пустая строка"
```

  - `undefined` — в `false`.

Операторы сравнения (см. *разд. 1.5*) при сравнивании значений разных типов предварительно преобразуют их к одному типу:

```
> 2 == 2
< true

> 2 != '2'
< false
```

Операторы `===` (строго равно) и `!==` (строго не равно) преобразования к единому типу не выполняют и в случае несовпадения типов операндов всегда возвращают `false` и `true` соответственно:

```
> 2 === 2
< true
```

```
> 2 === '2'
< false

> 2 !== 2
< false

> 2 !== '2'
< true
```

Для явного преобразования строки в число можно использовать оператор *унарный плюс*: `+`. Полученное в результате преобразования число возвращается в качестве результата:

```
> +'123'
< 123

> +'123.456'
< 123.456
```

Для преобразования строк в числа можно использовать языковые конструкции (это встроенные функции, которые мы рассмотрим в *уроке 3*):

- `parseInt(<строка>)` — преобразует значение *строка* в целое число, которое возвращается в качестве результата;
- `parseFloat(<строка>)` — преобразует значение *строка* в дробное число, возвращаемое в качестве результата.

Они корректно преобразуют в число строки, содержащие пробелы в начале и (или) конце, и те, в которых после цифр находятся буквы:

```
> parseInt('100')
< 100

> parseFloat('123.456')
< 123.456

> parseInt('   100   ')
< 100

> parseInt('100сто')
< 100

> parseFloat('123.456abcdef')
< 123.456
```

1.12. Самостоятельные упражнения

- Даны величины размера 17, 19 и 25 дюймов. Переведите их в сантиметры (для этого достаточно умножить величину в дюймах на 2,54).
- Проверьте, делится ли число 123456789 на 3 без остатка (*подсказка*: воспользуйтесь оператором взятия остатка от деления `%` и проверьте, равен ли получившийся остаток 0).

- Объявите переменную `sum2` и присвойте ей сумму чисел 500, 600 и 700.
- Объявите переменную `count` и присвойте ей число 3.
- Разделите значение переменной `sum2` на значение переменной `count`. Получившееся частное присвойте константе `average`. В результате у вас получится среднее арифметическое чисел 500, 600 и 700.
- Выведите полученное среднее арифметическое в консоли в формате:

  ```
  Среднее арифметическое: <значение>
  ```

- Проверьте, были ли ранее объявлены переменные `sum`, `count2` и `specialValue`.
- Вычислите квадратные корни от 2, 3, 4 и 7. Для этого воспользуйтесь следующей языковой конструкцией (подробности о ней — в *уроке 4*):

  ```
  Math.sqrt(<число, от которого нужно взять корень>)
  ```

  Извлеченный корень возвращается в качестве результата.

Урок 2
Выражения и управляющие конструкции

Веб-сценарии
Выражения
Условные выражения
Блоки
Выражения выбора
Циклы
Комментарии

2.1. Упражнение. Пишем первый веб-сценарий

Напишем наш первый веб-сценарий, который будет пересчитывать введенную пользователем величину размера из дюймов в сантиметры и выводить результат на экран.

> *Веб-сценарий* представляет собой набор инструкций, которые веб-обозреватель должен последовательно выполнить для достижения нужного результата. Такая инструкция называется *выражением*.

> Каждое выражение, записываемое в веб-сценарии, обязательно должно завершаться символом `;` (точка с запятой).

Для простоты сделаем наш сценарий внутренним, поместив его непосредственно в HTML-код небольшой учебной страницы.

> *Внутренний веб-сценарий* полностью записывается в HTML-коде веб-страницы.

Внутренний сценарий должен помещаться в парном теге `<script>`. У этого тега рекомендуется указать атрибут `type` со значением `text/javascript` — так мы сообщим веб-обозревателю, что сценарий написан на JavaScript.

1. В папке 2\!sources сопровождающего книгу электронного архива (см. *приложение 3*) найдем файл 2.1.html и скопируем его в какую-либо папку на локальном диске.

   Этот файл хранит веб-страницу, содержащую лишь заголовок первого уровня (тег `<h1>`) с текстом «Пересчет величин».

2. Откроем копию файла 2.1.html в текстовом редакторе и вставим сразу после тега <h1> пустой тег <script> (здесь и далее в тексте книги подобные вставки выделяются полужирным шрифтом), в который совсем скоро поместим код сценария:

```
<h1>Пересчет величин</h1>
<script type="text/javascript">
</script>
```

3. Сначала нам нужно вывести на экран стандартное диалоговое окно с полем ввода, в которое пользователь будет заносить требуемую величину в дюймах.

   Для этого вставим в тег <script> первое выражение сценария, которое выведет на экран такое окно и присвоит введенную пользователем величину константе ins:

```
<script type="text/javascript">
    const ins = window.prompt('Величина в дюймах', 0);
</script>
```

   Для вывода окна мы применили языковую конструкцию window.prompt() (подробнее о ней рассказано в *уроке 11*). В круглых скобках мы записали строку с пояснением, которое будет отображено в окне над полем ввода, и, через запятую, изначальное значение, что будет присутствовать в поле (у нас — 0). В качестве результата языковая конструкция вернет введенное в окно значение в виде строки.

   Поскольку значение переменной ins у нас после первого присваивания не меняется, мы объявили ее как константу — оператором const (об объявлении переменных разговор шел в *разд. 1.7*).

4. Добавим второе выражение, которое пересчитает полученную ранее дюймовую величину в сантиметры, умножив ее на 2,54, и присвоит результат константе cents:

```
<script type="text/javascript">
    const ins = window.prompt('Величина в дюймах', 0);
    const cents = ins * 2.54;
</script>
```

   Величину в дюймах, хранящуюся в переменной ins в виде строки, JavaScript сам преобразует в число перед тем, как выполнить умножение (подробнее о преобразовании типов говорилось в *разд. 1.11*).

5. Осталось вывести полученный «сантиметровый» результат на веб-страницу. Для этого добавим в сценарий последнее выражение, которое и выполнит вывод:

```
<script type="text/javascript">
    const ins = window.prompt('Величина в дюймах', 0);
    const cents = ins * 2.54;
    window.document.write('<p>', cents, '</p>');
</script>
```

Языковая конструкция `window.document.write()` (разговор о ней пойдет в *уроке 7*) выведет на страницу значения, что приведены через запятую в круглых скобках. Вывести можно как обычные величины, так и фрагменты HTML-кода, формирующие какие-либо новые элементы страницы. Здесь же мы вывели посредством этой конструкции открывающий тег `<p>`, рассчитанную величину в сантиметрах и закрывающий тег `</p>`, в результате чего на странице появится новый абзац.

Откроем страницу 2.1.html в веб-обозревателе — и на экране сразу же появится диалоговое окно для ввода значения (рис. 2.1).

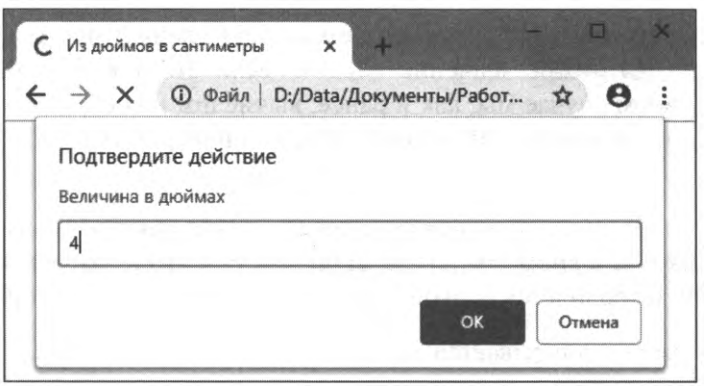


Рис. 2.1. Диалоговое окно для ввода значения

Введем в поле число 4, нажмем кнопку **ОК** — и на странице под заголовком **Пересчет величин** появится абзац с вычисленной величиной в сантиметрах (рис. 2.2).

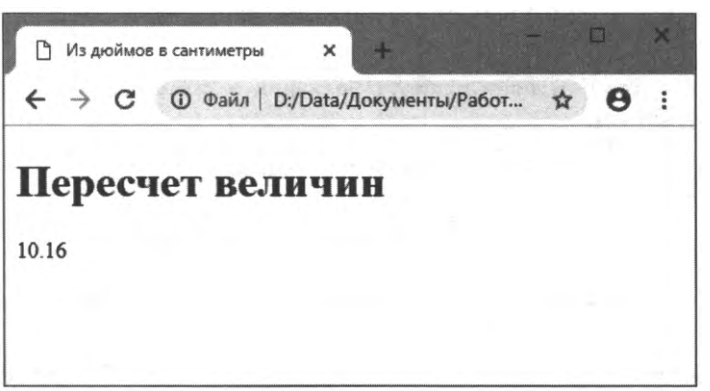


Рис. 2.2. Новый абзац с вычисленной величиной в сантиметрах

Попробуем пересчитать в сантиметры другую «дюймовую» величину. Для этого обновим открытую страницу, после чего на экране вновь появится диалоговое окно, в котором мы сможем ввести новое значение в дюймах.

2.2. Упражнение. Применяем условные выражения

У сценария, написанного в *разд. 2.1*, есть недостаток — если в диалоговом окне ввода величины нажать кнопку **Отмена**, на страницу будет выведено число 0. Дело в том, что по нажатию кнопки **Отмена** языковая конструкция `window.prompt` вернет значение `null`. Следующее выражение нашего сценария умножит это значение на 2,54, в результате чего и получится 0.

Давайте сделаем так, чтобы вместо нуля выводился осмысленный текст — например: «Величина не была введена».

Значит, в сценарии нам перед умножением на 2,54 следует проверить, вернула ли конструкция `window.prompt` значение `null`, и, если это так, вывести текст сообщения. В противном случае мы, как и ранее, вычислим на основе введенного числа «сантиметровую» величину. Выполнить такую проверку нам поможет условное выражение.

> *Условное выражение* включает в себя условие (аналогичное тому, что применяется в условном операторе из *разд. 1.9*) и два выражения. Если условие истинно, выполняется первое выражение, если ложно — второе.

Условное выражение записывается в формате:

```
if (<условие>)
    <выражение true>
else
    <выражение false>
```

Если *условие* истинно, выполняется *выражение true*, если ложно — *выражение false*.

Перепишем наш сценарий следующим образом (рис. 2.3).

```
const ins = window.prompt('Величина в дюймах', 0);
let cents;
if (ins == null)
    cents = 'Величина не была введена';
else
    cents = ins * 2.54;
window.document.write('<p>', cents, '</p>');
```

Рис. 2.3. В сценарий добавлено условное выражение

Условное выражение (на рис. 2.3 оно выделено рамкой) проверяет, равно ли значение константы `ins` (в которой сохраняется занесенная пользователем «дюймовая» величина) значению `null`. Если это так, в переменную `cents` заносится текст сообщения, в противном случае — вычисленная величина в сантиметрах.

Поскольку переменной `cents` значение присваивается не в момент объявления, мы объявили ее как обычную переменную — оператором `let`.

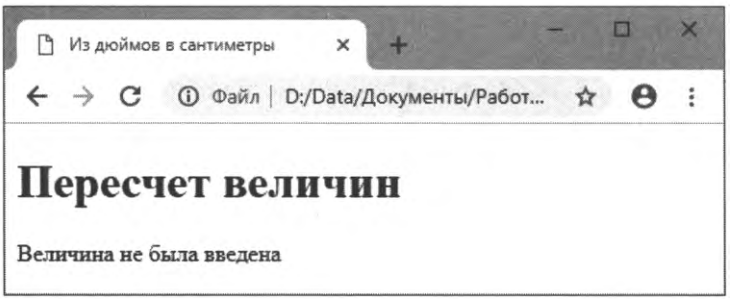


Рис. 2.4. Результат работы условного выражения по нажатию кнопки **Отмена**

Откроем страницу 2.1.html в веб-обозревателе, нажмем в диалоговом окне ввода значения кнопку **Отмена** и посмотрим на результат (рис. 2.4).

2.2.1. Сокращенная форма условного выражения

Условное выражение можно записать в *сокращенной форме*:

```
if (<условие>)
    <выражение true>
```

> Если *условие* истинно, выполняется *выражение true*, если ложно — ничего не произойдет.

Вот вариант сценария, использующего сокращенную форму условного выражения:

```
const ins = window.prompt('Величина в дюймах', 0);
let cents = 'Величина не была введена';
if (ins != null)
    cents = ins * 2.54;
window.document.write('<p>', cents, '</p>');
```

Сценарий получился даже несколько компактнее и теперь занимает 5 строк, а не 7, как ранее.

2.2.2. Множественное условное выражение

Множественное условное выражение представляет собой объединение произвольного количества обычных условных выражений. Оно записывается в формате:

```
if (<условие 1>)
    <выражение true 1>
else if (<условие 2>)
    <выражение true 2>
. . .
else if (<условие n-1>)
    <выражение true n-1>
else if (<условие n>)
    <выражение true n>
[else
    <выражение false>]
```

> Если *условие 1* истинно, выполняется *выражение true 1*, если истинно *условие 2* — *выражение true 2* и т. д. Если все *условия* ложны, будет выполнено *выражение false* в случае, когда оно присутствует в коде. Количество *условий* не ограничено.

Вот вариант сценария, который, помимо всего прочего, предупреждает о случайном вводе отрицательной величины:

```
const ins = window.prompt('Величина в дюймах', 0);
let cents;
if (ins == null)
    cents = 'Величина не была введена';
else if (ins < 0)
    cents = 'Величина не может быть отрицательной';
else
    cents = ins * 2.54;
window.document.write('<p>', cents, '</p>');
```

2.3. Упражнение. Используем блоки

Посмотрим на сценарий из *разд. 2.2*. Переменная `cents` фактически нужна только в том случае, если выполняется вычисление величины в сантиметрах. Поэтому и объявлять ее имеет смысл лишь тогда, когда требуется выполнить пересчет, — в выражении `true` (так мы сэкономим память). Если же пользователь нажал кнопку **Отмена**, мы можем просто вывести текст сообщения посредством конструкции `window.document.write`.

Но вычисление величины в сантиметрах и ее вывод — это два выражения. Однако условное выражение позволяет использовать только единичные выражения: `true` и `false`. Что делать? Применить блок.

> *Блок* (или *блочное выражение*) включает в себя произвольное количество единичных выражений.

Блок создается фигурными скобками `{` и `}`, в которые помещаются выражения, входящие в его состав.

Перепишем сценарий следующим образом:

```
const ins = window.prompt('Величина в дюймах', 0);
if (ins == null)
    window.document.write('<p>Величина не была ' +
        'введена</p>');
else {
    const cents = ins * 2.54;
    window.document.write('<p>', cents, '</p>');
}
```

Здесь в качестве выражения `else` мы применили блок, содержащий два выражения.

2.3.1. Переменные в блоках

В блоках можно объявлять переменные (что, собственно, и было продемонстрировано только что). При этом:

- переменные, объявленные операторами `let` и `const`, существуют лишь внутри блока. Как только выполнится последнее выражение блока, эти переменные удаляются из памяти:

```
else {
    let cents = ins * 2.54;
    window.document.write('<p>', cents, '</p>');
}
let cents2 = cents;
let message = 'Ошибка! Переменная cents уже не существует.';
```

- переменные, объявленные оператором `var`, остаются существовать и по завершении выполнения блока. В некоторых случаях это может оказаться полезным:

```
else {
    var cents = ins * 2.54;
    window.document.write('<p>', cents, '</p>');
}
let cents2 = cents;
let message = 'Переменная cents все еще существует, ' +
    'и к ней можно обратиться.';
```

2.4. Выражение выбора

Выражение выбора последовательно сравнивает заданное значение с набором указанных величин и при совпадении с какой-либо величиной выполняет соответствующий фрагмент кода.

Выражение выбора проще рассмотреть, сравнивая его с множественным условным выражением, при исполнении которого будет достигнут аналогичный результат.

Формат выражения выбора:

```
switch(<значение>) {
    case <величина 1>:
        <фрагмент 1>
        break;
    case <величина 2>:
        <фрагмент 2>
        break;
    . . .
    case <величина n-1>:
        <фрагмент n-1>
        break;
```

⇩

Аналогичное условное выражение:

```
if (<значение> === <величина 1>)
    <фрагмент 1>
else if (<значен.> ==== <вел. 2>)
    <фрагмент 2>
. . .
else if (<знач.> ==== <вел. n-1>)
    <фрагмент n-1>
else if (<знач.> ==== <велич. n>)
    <фрагмент n>
else
    <фрагмент default>
```

```
    case <величина n>:
        <фрагмент n>
        break;
    [default:
        <фрагм. default>]
}
```

> Несмотря на то, что выражение выбора может оказаться более громоздким, однако выполняется оно быстрее аналогичного множественного условного выражения.

Рассмотрим пример (предполагается, что переменная `screenSize` уже объявлена и хранит значение размера экрана в дюймах):

```
let s;
switch(screenSize) {
    case 3:
        s = 'Маленький экран';
        break;
    case 4:
        s = 'Маленький экран';
        break;
    case 5:
        s = 'Экран средних размеров';
        break;
    case 6:
        s = 'Большой экран';
        break;
    default:
        s = 'Таких экранов в телефонах не бывает';
}
```

Здесь, если в переменной `screenSize` записано число `4`, будет выполнен второй по счету фрагмент кода (записанный после строки `case 4:`), и переменная `s` в качестве значения получит строку `'Маленький экран'`.

При использовании выражений выбора следует иметь в виду два момента:

- при сравнивании заданного *значения* с *величинами* JavaScript применяет оператор `===` (строго равно). Это значит, что преобразование типов при сравнении не проводится.

  В нашем случае, если в переменной `screenSize` записана строка `'4'`, она не совпадет ни с одной из четырех величин, присутствующих в этом выражении выбора. В результате будет выполнен фрагмент `default`, и в переменной `s` окажется строка `'Таких экранов в телефонах не бывает'`;

- в конце каждого *фрагмента* (за исключением самого последнего) ставится *оператор прерывания* `break`. Он немедленно прерывает выполнение выражения выбора.

Если эти операторы опустить, после выполнения *фрагмента*, соответствующего совпавшей *величине*, станут выполняться последующие *фрагменты*, пока не будет достигнут конец выражения выбора. Понятно, что результат исполнения кода будет не тем, на который мы рассчитывали.

Однако эту особенность выражения выбора можно использовать на практике, чтобы сократить код.

Вот как это можно сделать на приведенном ранее примере (исправленный код выделен полужирным шрифтом):

```
let s;
switch(screenSize) {
    case 3:
    case 4:
        s = 'Маленький экран';
        break;
    case 5:
        s = 'Экран средних размеров';
        break;
    case 6:
        s = 'Большой экран';
        break;
    default:
        s = 'Таких экранов в телефонах не бывает';
}
```

При совпадении значения переменной `screenSize` с величинами `3` и `4` будет выполнен один и тот же фрагмент кода — присваивающий переменной `s` строку `'Маленький экран'`.

2.5. Упражнение. Используем цикл со счетчиком

Напишем веб-сценарий, который выведет на экран таблицу с числами от 1 до 10 и квадратными корнями из них. Чтобы не писать 10 однотипных выражений, используем цикл со счетчиком.

Цикл — фрагмент кода, выполняющийся многократно.

Условный цикл — выполняется, пока истинно указанное условие.

Цикл со счетчиком — выполняется заданное количество раз.

Счетчик цикла — переменная, хранящая количество уже прошедших выполнений (*проходов*, или *итераций*) цикла.

Цикл со счетчиком записывается в формате:

```
for ([<установка>]; [<условие>]; [<приращение>])
    <тело цикла>
```

Здесь:

- *установка* — выражение, присваивающее счетчику начальное значение;
- *условие* — пока возвращает в качестве результата `true`, цикл выполняется, но как только вернет `false`, исполнение цикла завершается. Практически всегда *условие* проверяет, не превышает ли текущее значение счетчика величину необходимого количества проходов цикла;
- *приращение* — выражение, изменяющее значение счетчика;
- *тело цикла* — единичное выражение или блок, который и будет выполняться заданное количество раз. Собственно, *тело* — это код, для которого пишется цикл.

* * *

1. В папке 2\!sources сопровождающего книгу электронного архива (см. *приложение 3*) найдем файл 2.5.html и скопируем его в какую-либо папку на локальном диске.

   Этот файл хранит веб-страницу, содержащую заголовок первого уровня с текстом «Пересчет величин» и парный тег `<table>`, создающий «пустую» таблицу.

2. Откроем копию файла 2.5.html в текстовом редакторе и вставим между тегами `<table>` и `</table>` код, объявляющий переменную-счетчик `i` и создающий сам цикл со счетчиком:

```
<script type="text/javascript">
    let i;
    for (i = 1; i <= 10; ++i) {
    }
</script>
```

   В нашем цикле:

   - *установка* (`i = 1`) — присваивает счетчику значение `1` (первое число, квадратный корень которого надо вычислить);
   - *условие* (`i <= 10`) — проверяет, не превышает ли значение счетчика `10` (количество проходов цикла и одновременно последнее число, чей квадратный корень нужно вычислить);
   - *приращение* (`++i`) — инкрементирует (увеличивает на `1`) значение счетчика.

3. Напишем *тело цикла* — выражения, вычисляющие квадратные корни (выделены полужирным шрифтом):

```
for (i = 1; i <= 10; ++i) {
    window.document.write('<tr><td>', i, '</td>');
    window.document.write('<td>', Math.sqrt(i), '</td></tr>');
}
```

   На каждом проходе цикла мы создаем строку таблицы с двумя ячейками: в левой выводится изначальное число, а в правой — квадратный корень из него. Для вычисления квадратного корня от *числа* мы применили конструкцию `Math.sqrt(<число>)`, упомянутую в *разд. 1.12*.

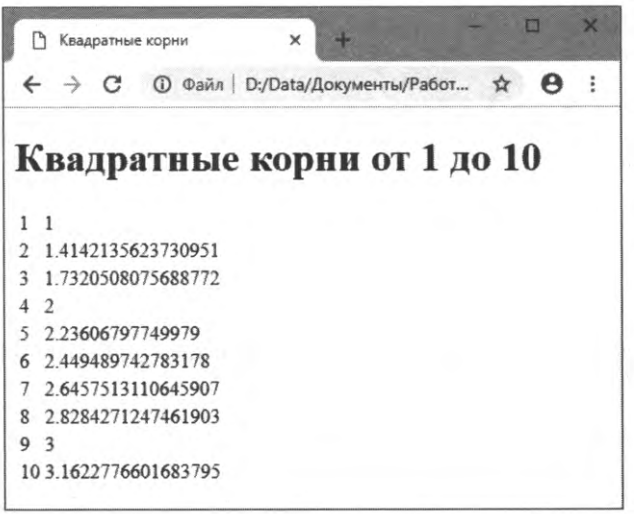


Рис. 2.5. Работа цикла, вычисляющего квадратные корни чисел от 1 до 10

Откроем страницу 2.5.html в веб-обозревателе и проверим, работает ли наш цикл (рис. 2.5).

- *Приращение* совершенно не обязано именно инкрементировать счетчик. Оно может его декрементировать — и наш цикл выведет квадратные корни чисел от 10 до 1:

```
for (i = 10; i >= 1; --i) {
    . . .
}
```

Оно может увеличивать его на 2, что может понадобиться для вывода квадратных корней от четных чисел:

```
for (i = 2; i <= 10; i += 2) { . . . }
```

- И *установка*, и *условие*, и *приращение* не являются обязательными для указания:
  - можно опустить *установку* — тогда начальное значение счетчику следует присвоить перед исполнением цикла:

```
let i = 1;
for (; i <= 10; ++i) {
    . . .
}
```

  - можно опустить *условие* — но тогда придется производить необходимую проверку в начале *тела цикла* и прерывать выполнение цикла оператором `break`, знакомым нам по *разд. 2.4*:

```
for (i = 1; ; ++i) {
    if (i > 10)
        break;
    . . .
}
```

- можно не указывать *приращение* — тогда его придется выполнять в конце *тела цикла*:

```
for (i = 1; i <= 10;) {
    . . .
    i++;
}
```

В последних двух случаях цикл со счетчиком становится похожим на циклы двух других разновидностей, которые мы скоро рассмотрим.

2.5.1. Объявление счетчика непосредственно в цикле

Объявление переменной-счетчика можно выполнить непосредственно в *установке* цикла. В этом случае счетчик будет существовать только в *теле цикла*:

```
for (let i = 1; i <= 10; ++i) {
    . . .
    // Здесь, в теле цикла, переменная-счетчик i существует
}
// А здесь, за пределами цикла, — уже нет
```

2.5.2. В приращении можно записать несколько выражений

В *приращении* цикла можно выполнить сразу несколько действий, записав необходимые выражения через запятую. В следующем примере одновременно выполняется инкремент значения переменной `i` и увеличение на 2 значения переменной `j`:

```
for (let i = 0, j = 0; i <= 10; ++i, j += 2) {
    . . .
    // Используем значения переменных i и j в вычислениях
}
```

2.6. Циклы с предусловием и постусловием

> *Цикл с предусловием* выполняется, пока заданное условие остается истинным. Условие проверяется *перед* каждым проходом цикла.

Формат цикла с предусловием:

```
while (<условие>)
    <тело цикла>
```

Пример:

```
let i = 1;
while (i <= 10) {
    window.document.write('<tr><td>', i, '</td>');
    window.document.write('<td>', Math.sqrt(i), '</td></tr>');
    i++;
}
```

Цикл с постусловием аналогичен циклу с предусловием, но условие проверяется *после* каждого прохода цикла. Такой цикл выполнится хотя бы один раз, даже если условие изначально ложно.

Формат цикла с постусловием:

```
do
    <тело цикла>
while (<условие>)
```

Пример:

```
let i = 1;
do {
    window.document.write('<tr><td>', i, '</td>');
    window.document.write('<td>', Math.sqrt(i), '</td></tr>');
    i++;
} while (i <= 10)
```

2.7. Операторы *break* и *continue*

Оператор прерывания `break` прерывает выполнение всего цикла (а также выражения выбора — см. *разд. 2.6*). Он не принимает аргументов и не возвращает результата.

Этот цикл выведет на экран квадратные корни чисел от `0` до `5`:

```
let i;
for (i = 1; i <= 10; ++i) {
    if (i > 5)
        break;
    window.document.write('<tr><td>', i, '</td>');
    window.document.write('<td>', Math.sqrt(i), '</td></tr>');
}
```

Оператор прерывания прохода `continue` прерывает выполнение текущего прохода цикла и начинает выполнение следующего. Не принимает аргументов и не возвращает результата.

Этот цикл выведет на экран только квадратные корни, являющиеся дробными числами:

```
let i, sq;
for (i = 1; i <= 10; ++i) {
    sq = Math.sqrt(i);
    if (sq == Math.floor(sq))
        continue;
    window.document.write('<tr><td>', i, '</td>');
    window.document.write('<td>', Math.sqrt(i), '</td></tr>');
}
```

Языковая конструкция `Math.floor(<число>)` округляет заданное *число* до ближайшего меньшего целого и возвращает результат. Целые же числа она «пропускает» через себя без изменения, и этим можно пользоваться, чтобы узнать, является ли число целым.

2.8. Комментарии

> *Комментарий* — фрагмент программного кода, не обрабатываемый веб-обозревателем. Служит для размещения в коде всевозможных заметок, пояснений, напоминаний и др.

JavaScript поддерживает два вида комментариев:

- однострочный — начинается с последовательности символов `//` (два слеша):

```
// Проверяем, хранит ли переменная sq целое число
if (sq == Math.floor(sq))
    . . .
```

- многострочный — начинается с последовательности символов `/*` и заканчивается последовательностью `*/`:

```
/*
    Объявляем переменные sum, count
    и константу convert
*/
let sum, count;
const convert = 2.54;
```

2.9. Строгий режим

> Работая в *строгом режиме*, веб-обозреватель требует, чтобы все используемые в сценарии переменные были предварительно объявлены. Попытка присвоить значение необъявленной переменной в этом случае вызовет ошибку.

Чтобы переключить веб-обозреватель в строгий режим, достаточно самым первым выражением в сценарии поставить строку `'use strict'`:

```
'use strict';
/*
    Строгий режим включен!
    Не забываем объявлять все переменные!
*/
let sum, count;
. . .
```

2.10. Как набирать JavaScript-код?

Каждое выражение JavaScript обычно записывается на отдельной строке:

```
const ins = window.prompt('Величина в дюймах', 0);
const cents = ins * 2.54;
window.document.write('<p>', cents, '</p>');
```

Короткие выражения можно записать в одну строку:

```
let sum, count; const convert = 2.54;
```

Между отдельными значениями, именами переменных, операторами, ключевыми словами и прочими языковыми конструкциями обычно ставятся необязательные пробелы — так код лучше читается:

```
const cents = ins * 2.54;
```

Но их можно не ставить — ради компактности:

```
const cents=ins*2.54;
```

Так часто удается значительно уменьшить объем кода.

Вложенность выражений в блоки и другие более сложные выражения отмечается отступами слева (обычно в 4 пробела):

```
if (ins != null)
    cents = ins * 2.54;

for (i = 1; i <= 10; ++i) {
    window.document.write('<tr><td>', i, '</td>');
    window.document.write( . . . );
}
```

Слишком длинные выражения можно переносить по строкам, разрывая строки в промежутках между языковыми конструкциями. Строки, на которых находятся оставшиеся части выражения, часто выделяются увеличенными отступами слева:

```
window.document.write('<td><strong><em>', Math.sqrt(i),
    '</em></strong></td></tr>');
```

2.11. Самостоятельные упражнения

- Доработайте сценарий, переводящий величины из дюймов в сантиметры, таким образом, чтобы он выводил результат в двух абзацах: в первом — величину в дюймах, а во втором — в сантиметрах.
- Дополните выводимую величину в дюймах обозначением единицы измерения: «дюйм», «дюйма» или «дюймов». Для округления числа используйте языковую конструкцию `Math.floor(`*<число>*`)`, а для выбора нужного обозначения единицы измерения — выражение выбора. Заодно дополните вывод в сантиметрах обозначением единицы измерения — «см.».

  У вас должно получиться так (рис. 2.6).

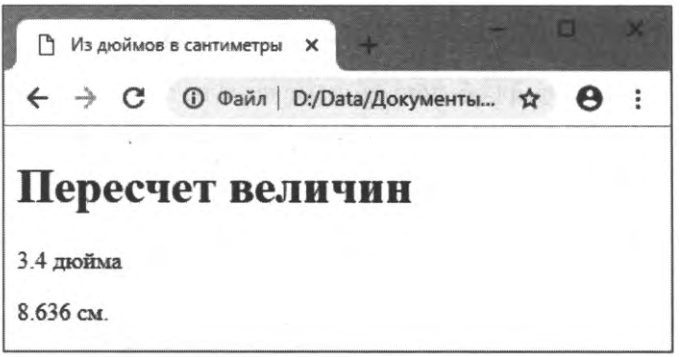


Рис. 2.6. Ожидаемый результат доработки сценария

- Напишите страницу 2.9.1.html с веб-сценарием, который выведет квадратные корни от чисел 10, 20 . . . 200. Используйте для этого цикл со счетчиком.
- Напишите страницу 2.9.2.html со сценарием, который будет запрашивать в цикле числа, суммировать их, а при вводе числа `0` — выводить на экран среднее арифметическое введенных чисел.

Урок 3
Функции и массивы

Функции, их объявление и вызов
Локальные переменные и вложенные функции
Функции с необязательными параметрами
Функции с произвольным количеством параметров
Рекурсия
Анонимные функции
Функции-стрелки
Внешние веб-сценарии
Массивы

3.1. Упражнение. Используем функции

Напишем веб-сценарий, который выведет на экран таблицу со значениями длин окружностей диаметром 2, 5, 10 и 20 м. Поскольку нам придется производить одни и те же операции с разными значениями в разных местах кода, для их выполнения мы используем функцию.

> *Функция* — фрагмент программного кода, к которому можно обратиться из другого места сценария. Может принимать произвольное количество параметров и возвращать результат их обработки.

> *Параметр* — значение, переданное функции для выполнения вычислений.

Функция должна иметь уникальное *имя*, по которому выполняется обращение к ней. Именование функций выполняется по тем же правилам, что и именование переменных (см. *разд. 1.7*).

Дадим нашей функции имя `circleLength`.

1. Взяв за образец веб-страницу 2.5.html из папки 2\!sources сопровождающего книгу электронного архива (см. *приложение 3*), напишем страницу 3.1.html с заголовком первого уровня, «пустым» тегом `<table>` и находящимся в нем «пустым» тегом `<script>`, в который поместим наш сценарий.

   Перед первым использованием функции необходимо выполнить ее объявление.

   > *Объявление функции* — ее описание, включая имена и состав принимаемых параметров.

Объявление функции записывается в формате:

```
function <имя функции>([<список имен параметров через запятую>])
    <тело функции в виде блока>
```

> *Тело функции* — ее исполняемый код — всегда оформляется в виде блока, даже если и представляет собой единичное выражение.

В круглых скобках приводятся имена принимаемых функцией параметров. При выполнении функции на их основе будут созданы доступные только в теле функции переменные, которым будут присвоены реальные значения параметров.

> Объявление функции обязательно должно располагаться перед первым обращением к ней (ее вызовом).

2. В теге `<script>` напишем код объявления функции `circleLength`, принимающей один параметр — диаметр окружности с именем `d`:

```
function circleLength(d) {
    let l;
    l = d * Math.PI;
    return l;
}
```

В теле функции мы объявили переменную `l`, присвоили ей результат умножения диаметра `d` на число π и указали вернуть его из функции в качестве результата — длины окружности.

> *Оператор возврата результата* `return` выполняет возврат указанного значения в качестве результата выполнения функции. Он записывается в формате:
>
> `return <возвращаемое значение>`

Если оператор `return` в теле функции отсутствует, функция не будет возвращать результата.

Для получения значения числа π мы применили языковую конструкцию `Math.PI`, рассказ о которой пойдет в *уроке 4*.

3. Добавим код, вычисляющий длину окружности диаметром 2 м и выводящий ее на экран:

```
window.document.write('<tr><td>2</td>');
window.document.write('<td>', circleLength(2), '</td></tr>');
```

Здесь мы выполнили вызов функции `circleLength`, передав ей в качестве значения единственного параметра число 2 (код вызова выделен полужирным шрифтом).

> *Вызов функции* — указание выполнить объявленную ранее функцию, возможно, с передачей ей параметров и сохранением возвращенного результата.

Вызов функции записывается в формате:

```
<имя функции>([<список значений параметров через запятую>])
```

> *Оператор вызова функции* `()` (пара круглых скобок) ставится после имени функции, а внутри него через запятую приводятся значения передаваемых функции параметров. Если функция не принимает параметров, в скобках не записывается ничего.

4. Допишем код, вычисляющий и выводящий длины окружностей диаметром 5, 10 и 20 м:

```
window.document.write('<tr><td>5</td>');
window.document.write('<td>', circleLength(5), '</td></tr>');
window.document.write('<tr><td>10</td>');
window.document.write('<td>', circleLength(10), '</td></tr>');
window.document.write('<tr><td>20</td>');
window.document.write('<td>', circleLength(20), '</td></tr>');
```

Открыв страницу 3.1.html в веб-обозревателе, мы увидим таблицу со значениями длин окружностей разных диаметров (рис. 3.1).

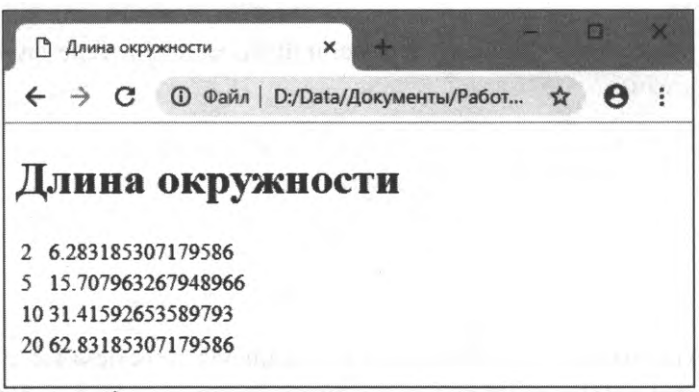


Рис. 3.1. Результат вычисления длин окружностей разных диаметров

Приведем еще несколько примеров функций:

- функция, принимающая три параметра:

```
function computeSomeValue(a, b, c) {
    . . .
}
. . .
let result = computeSomeValue(1, 2, 3);
```

- функция, не принимающая параметров и не возвращающая результата:

```
function doSomething() {
    . . .
}
. . .
doSomething();
```

- вычисляя возвращаемый результат непосредственно в операторе `return`, мы можем заметно сократить код:

```
function circleLength(d) { return d * Math.PI; }
```

3.2. Локальные переменные и вложенные функции

В примере из *разд. 3.1* было показано, что в теле функции можно объявлять переменные.

> *Локальная переменная* — переменная, объявленная в теле функции. Доступна только внутри тела функции и после ее выполнения уничтожается. Это касается переменных, объявленных любым оператором: `let`, `const` или `var` (подробнее об объявлении переменных рассказывалось в *разд. 1.7*).

В теле функции можно обращаться к *глобальным переменным* — объявленным в обычном коде.

Попытка обратиться к глобальной переменной `globalVar` в теле функции `someFunc` увенчается успехом:

```
let globalVar = 123;
function someFunc( . . . ) {
    . . .
    let localVar = globalVar;
    . . .
}
```

> Если существуют и глобальная, и локальная переменные с одинаковыми именами, попытка обратиться по этому имени в теле функции вызовет обращение к *локальной переменной*.

Так, если мы обратимся по имени `someVar` к переменной, объявленной в теле функции `someFunc`, то получим значение локальной переменной `someVar`:

```
let someVar = 123;
function someFunc( . . . ) {
    let someVar = 321;
    . . .
    let otherVar = someVar;  // 321
    . . .
}
```

> *Вложенная функция* — функция, объявленная в теле другой функции. Доступна только в теле функции, в которой объявлена. Имеет доступ к локальным переменным, объявленным во «внешней» функции.

Пример:

```
// Объявление "внешней" функции
function outerFunc(a, b, c) {
    let y = 0;
    // Объявление вложенной функции
    function innerFunc(d, e) {
        . . .
        // Во вложенной функции доступны локальные переменные
        // из "внешней" функции
        let temp = y;
        . . .
    }
    . . .
    // Вызов вложенной функции
    innerFunc(a, 6);
    . . .
}
```

3.3. Функции с необязательными параметрами

> *Необязательный параметр* функции при вызове функции можно не указывать — в этом случае он получит значение по умолчанию, заданное в объявлении функции.

Чтобы сделать параметр функции необязательным, достаточно в объявлении функции присвоить ему нужное значение по умолчанию. Это выполняется обычным оператором присваивания =.

Необязательные параметры обязательно *должны располагаться в самом конце списка принимаемых функцией параметров*. В противном случае мы получим ошибку.

Рассмотрим вариант функции `circleLength` с необязательным параметром, имеющим значение по умолчанию 2:

```
function circleLength(d = 2) {
    let l;
    l = d * Math.PI;
    return l;
}
```

Если вызвать эту функцию без параметров, параметр по умолчанию получит значение 2, и функция успешно вернет результат:

```
circleLength()  // 6.283185307179586
```

Необязательных параметров может быть сколько угодно:

```
function someFunc(a, b, c, d = 1, e = 'Пример') {
    . . .
}
```

3.3.1. Функции с необязательными параметрами в «старом» стиле

Объявление в функциях необязательных параметров оператором = стало возможным только в последней версии JavaScript. В предыдущих версиях оно не поддерживалось, и для этого применялся другой подход. Рассмотрим его — на тот случай, если нам придется переписывать старый код.

У JavaScript есть особенность: если при вызове функции указать меньшее количество параметров, чем присутствует в ее объявлении, то не указанным в вызове параметрам будет присвоено значение `undefined`. Тогда в теле функции можно выполнить проверку, хранит ли параметр значение `undefined`, и, если так, присвоить ему нужное значение по умолчанию. Для этого можно применить условный оператор (см. *разд. 1.9*).

Для практики перепишем функцию `circleLength` под старые версии JavaScript:

```
function circleLength(d) {
    d = (d === undefined) ? 2 : d;
    let l;
    l = d * Math.PI;
    return l;
}
```

Однако чаще для проверки такого рода использовали оператор логического ИЛИ `||`. У него тоже есть любопытная особенность. Если оба его операнда не являются логическими величинами, оператор преобразует первый операнд к логическому типу. Если в результате получится `true`, оператор `||` вернет *изначальное значение первого операнда*, если `false` — *изначальное значение второго операнда*.

Исправим функцию, применив оператор логического ИЛИ:

```
function circleLength(d) {
    d = d || 2;
    . . .
}
```

3.4. Функции с произвольным количеством параметров

JavaScript позволяет достаточно просто объявить функцию, принимающую произвольное количество параметров. Для этого нужно:

- объявить функцию, не принимающую параметров;
- в теле функции получить количество принятых параметров, обратившись к языковой конструкции `arguments.length`;
- получить значения принятых параметров посредством языковой конструкции: `arguments[<порядковый номер параметра>]`.

  Только нужно иметь в виду, что нумерация параметров начинается с `0`, а не с `1`.

Более подробно об этих языковых конструкциях мы поговорим в *уроке 4*.

Вот функция, принимающая произвольное количество параметров и формирующая на их основе маркированный список:

```
function listValues() {
    let l = arguments.length, i;
    window.document.write('<ul>');
    for (i = 0; i < l; i++)
        window.document.write('<li>', arguments[i], '</li>');
    window.document.write('</ul>');
}
```

Вызовем ее со следующими параметрами:

```
listValues('HTML', 'CSS', 'JavaScript');
```

И получим такой результат (рис. 3.2).

Рис. 3.2. Маркированный список на основе вызова функции с произвольным количеством параметров

3.5. Упражнение. Используем рекурсию

Напишем веб-сценарий, запрашивающий у пользователя число, вычисляющий его факториал и выводящий полученный результат на страницу. Для этого мы используем рекурсивную функцию, которая вызывает саму себя.

> *Рекурсия* — вызов функцией самой себя.

1. Пересохраним страницу 2\ex2.1\2.1.html под именем 3.5.html, откроем ее в текстовом редакторе, исправим текст названия (которое записывается в теге `<title>`) и заголовка первого уровня на «Факториал» и удалим из тега `<script>` весь JavaScript-код.
2. Впишем в тег `<script>` код сценария, вычисляющего факториал:

```
function factorial(val) {
    if (val == 1)
        return 1;
    else
        return val * factorial(val - 1);
}
const a = window.prompt('Введите число', 1);
window.document.write('<p>', factorial(a), '</p>');
```

Вычисление факториала выполняет функция `factorial`. Если в качестве параметра она получит `1`, то вернет значение факториала от единицы, также равное `1`.

В противном случае она умножит полученное число на факториал числа, меньшего на единицу, который получит, вызвав себя. Это и есть рекурсия в нашем случае.

Откроем страницу 3.5.html в веб-обозревателе, введем число `5` в открывшееся окно задания значения и посмотрим, правильно ли вычислен факториал этого числа (рис. 3.3).

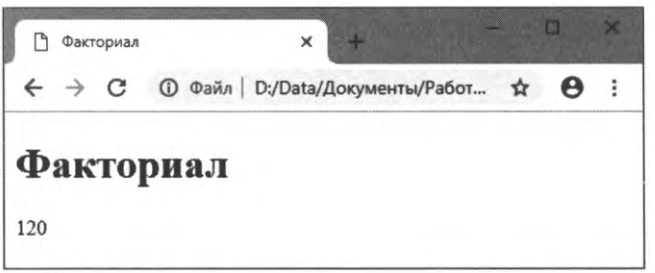


Рис. 3.3. Вычисление факториала числа 5

> *Бесконечная рекурсия* — случай, когда функция вызывает саму себя бесконечно.

При каждом вызове функции из тела другой функции веб-обозреватель заносит сведения об этом вызове в особую область памяти, называемую *стеком вызовов*. Эта область имеет ограниченный объем — несколько тысяч позиций, и по ее исчерпанию выполнение сценария будет прервано с выдачей сообщения об ошибке.

> Бесконечная рекурсия — это плохо!

В написанном нами сценарии череда последовательных вызовов функцией `factorial` самой себя прервется, как только при очередном вызове она получит в качестве параметра число 1.

3.6. Анонимные функции и функции-стрелки

> *Анонимная функция* — функция, не имеющая имени. Объявляется так же, как обычная (именованная), но без указания имени (формат объявления функции был приведен в *разд. 3.1*).

Объявляемая анонимная функция сразу же должна быть присвоена какой-либо переменной — в противном случае она будет потеряна.

Реализуем функцию `circleLength` как анонимную и присвоим ее переменной `anonFunc`:

```
const anonFunc = function (d) {
    let l;
    l = d * Math.PI;
    return l;
}
```

> Вызов анонимной функции выполняется указанием имени переменной, в которой она хранится, и оператора вызова функции `()`.

Вызываем анонимную функцию, обратившись к переменной `anonFunc`, в которой она хранится:

```
let cl = anonFunc(5);
```

> Анонимные функции можно передавать другим функциям в качестве параметров.

Объявляем функцию `computeSomeValue`, которая принимает анонимную функцию в качестве параметра:

```
function computeSomeValue(func) {
    . . .
    let l = func(10);
}
```

и вызываем, передав ей функцию из переменной `anonFunc` в качестве параметра:

```
let result = computeSomeValue(anonFunc);
```

> *Функция-стрелка* — разновидность анонимных функций, появившаяся в JavaScript версии ES6. Объявляется с применением последовательности символов =>, напоминающей стрелку, отчего и получила такое название.

Функция-стрелка может быть объявлена в двух форматах:

- полном:

  ```
  ([<список параметров через запятую>]) => <тело функции в виде блока>
  ```

  Пример:

  ```
  const anonFunc2 = (d) => {
      let l;
      l = d * Math.PI;
      return l;
  }
  ```

- сокращенном:

  ```
  ([<список параметров через запятую>]) => <единичное выражение>
  ```

  Значение, вычисленное *выражением*, будет возвращено в качестве результата.

  Пример:

  ```
  const anonFunc3 = (d) => d * Math.PI;
  ```

  В сокращенном формате можно записать только совсем простую функцию.

Функции-стрелки полностью аналогичны обычным анонимным функциям, но отнимают меньше системных ресурсов. Однако они имеют один недостаток, о котором мы узнаем в *уроке 15*.

3.7. Функциональный тип данных

Любая функция — неважно: именованная, анонимная или функция-стрелка — относится к *функциональному типу данных*. Оператор получения типа `typeof` (см. *разд. 1.10*) при вызове с операндом-функцией возвращает строку `'function'`:

```
let t = typeof circleLength  // 'function'
t = typeof anonFunc          // 'function'
t = typeof anonFunc2         // 'function'
t = typeof anonFunc3         // 'function'
```

3.8. Упражнение. Создаем внешний веб-сценарий

В *разд. 3.1* мы написали удачную функцию `circleLength` и теперь хотим использовать ее при разработке других сайтов, для чего вынесем объявление этой функции во внешний веб-сценарий.

> *Внешний веб-сценарий* — веб-сценарий, сохраненный в отдельном файле.

Файл с внешним сценарием (*файл веб-сценария*) представляет собой обычный текстовый файл. Он должен хранить только программный код JavaScript и иметь расширение js.

Для того чтобы веб-обозреватель смог загрузить и выполнить внешний сценарий, необходимо привязать его к веб-странице. Такая *привязка* выполняется все тем же тегом `<script>`, не имеющим содержимого, в атрибуте `src` которого записывается ссылка на файл сценария.

> *Библиотека* — внешний веб-сценарий, предназначенный для использования на разных веб-страницах, в том числе другими разработчиками.

Сохраним внешний сценарий с объявлением функции в файле 3.8.js.

1. Откроем страницу 3.1.html, найдем в ней фрагмент JavaScript-кода, объявляющий функцию `circleLength`:

   ```
   function circleLength(d) {
       . . .
   }
   ```

   выделим его и вырежем в буфер обмена.

2. Создадим в текстовом редакторе новый файл и вставим в него фрагмент из буфера обмена.

3. Сохраним этот файл в той же папке, где находится файл страницы 3.1.html, назначив ему имя 3.8.js и задав кодировку UTF-8.

4. Вернемся к странице и добавим в секцию заголовка (тег <head>) код, привязывающий файл сценария 3.8.js:

```
<head>
    . . .
    <script type="text/javascript" src="3.8.js"></script>
</head>
```

Проверим страницу в работе, открыв ее в веб-обозревателе. Если мы не допустили ошибок, то увидим таблицу со значениями длин окружностей разных диаметров.

3.9. Встроенные функции JavaScript

В JavaScript присутствует ряд *встроенных* (реализованных непосредственно в веб-обозревателе) функций. Вот они:

- `parseInt (<строка>)` — преобразует *строку* в целое число и возвращает его в качестве результата;
- `parseFloat(<строка>)` — преобразует *строку* в дробное число и возвращает его в качестве результата.

  Эти функции мы уже рассматривали в *разд. 1.11*;
- `isFinite(<число>)` — возвращает `true`, если *число* является конечным, и `false`, если это значение `Infinity`, `-Infinity` или `NaN`:

```
> isFinite(10)
< true
> isFinite(10 / 0)
< false
```

- `isNaN(<число>)` — возвращает `true`, если в качестве *числа* передано значение `NaN`, и `false` в противном случае;
- `eval(<строка с выражением>)` — вычисляет выражение, переданное в виде *строки*, и возвращает результат:

```
> eval('(10 + 20) / 2')
< 15
```

3.10. Массивы. Объектный тип данных

Массив — набор пронумерованных значений, хранящихся в одной переменной.

Элемент массива — отдельное значение, хранящееся в массиве. Может принадлежать любому типу, поддерживаемому JavaScript.

Индекс — порядковый номер элемента массива. Нумерация элементов массива начинается с `0`.

Размер массива — количество элементов, хранящихся в массиве.

Создать массив можно, указав все входящие в него элементы через запятую и взяв их в квадратные скобки. Созданный массив присваивается какой-либо переменной:

- создаем массив `lengths` и помещаем в него числа 2, 5, 10 и 20:

```
> let lengths
> lengths = [2, 5, 10, 20]
< (4) [2, 5, 10, 20]
```

  В качестве результата здесь возвращается значение вида:

```
(<размер массива>) [<элементы массива через запятую>]
```

- создаем массив `languages` и помещаем в него названия языков, применяемых в веб-разработке:

```
> let languages
> languages = ['HTML', 'CSS', 'JavaScript']
< (3) ["HTML", "CSS", "JavaScript"]
```

- можно создать и пустой массив, указав пустые квадратные скобки:

```
> let emptyArray = []
```

Получить доступ к элементу массива по его индексу можно, применив *оператор доступа к элементу массива* `[]` (пара квадратных скобок), который записывается в формате:

```
<переменная, в которой хранится массив>[<индекс>]
```

Можно как получать текущее значение из элемента массива, так и заносить в него новое значение:

- извлекаем второй (с индексом `1` — не забываем, что нумерация элементов начинается с `0`) элемент массива `lengths`:

```
> lengths[1]
< 5
```

- меняем значение третьего (индекс `2`) элемента того же массива на `15`:

```
> lengths[2] = 15
> lengths
< (4) [2, 5, 15, 20]
```

В массив можно добавлять новые элементы, использовав тот же синтаксис:

- добавляем еще один элемент в массив `languages` — четвертый по счету:

```
> languages[3] = 'PHP'
> languages
< (4) ["HTML", "CSS", "JavaScript", "PHP"]
```

- создаем в том же массиве новый элемент, дав ему индекс `7`:

```
> languages[7] = 'Python'
> languages
< (8) ["HTML", "CSS", "JavaScript", "PHP", empty × 3, "Python"]
```

Помимо элемента с индексом 7, будут созданы элементы с индексами 4, 5 и 6, чтобы заполнить образовавшуюся «прореху». Они получат значение undefined, о чем сообщает строка empty × 3, выведенная в консоли. Давайте убедимся в этом:

```
> languages[4]
< undefined
```

> *Вложенный массив* — массив, хранящийся в элементе другого массива (*внешнего*).

Создаем в пустом массиве emptyArray два элемента с вложенными массивами:

```
> emptyArray[0] = [1, 2, 3]
> emptyArray[1] = [4, 5, 6]
```

Для получения доступа к элементу вложенного массива оператор доступа к элементу [] следует записать дважды: первый оператор выполнит доступ к элементу внешнего массива, а второй — к элементу хранящегося в нем вложенного массива.

Извлекаем значение третьего элемента вложенного массива, хранящегося во втором элементе массива emptyArray:

```
> emptyArray[1][2]
< 6
```

Поскольку элементы массива имеют последовательно увеличивающиеся номера-индексы, массивы удобно обрабатывать в циклах. Для примера переберем элементы массива lengths и выведем их через пробел:

```
let i;
for (i = 0; i < 4; i++)
    window.document.write(lengths[i], ' ');
```

Массивы относятся к *объектному типу*. Оператор получения типа typeof (см. *разд. 1.10*) при вызове с операндом-массивом возвращает строку 'object':

```
> typeof languages
< "object"
```

3.11. Самостоятельные упражнения

- Напишите страницу 3.11.1.html с веб-сценарием, выводящим значения площадей кругов диаметром 10, 20 и 35 м. Для этого в коде сценария объявите функцию circleSquare, принимающую в качестве единственного параметра диаметр окружности и возвращающую вычисленную на его основе площадь круга (для ее вычисления используйте формулу πr^2, где r — радиус окружности).
- Напишите страницу 3.11.2.html со сценарием, содержащим объявление функции average, которая принимает произвольное количество параметров и вычисляет их среднее арифметическое (при этом параметры, не относящиеся к числовому

типу, пусть игнорируются). Проверьте функцию на следующих двух наборах параметров:

```
average(10, 20, 30, 40, 50, 60, 70, 80, 90, 100)
average(10, '20', 30, '40', 50, '60', 70, '80', 90, '100')
```

- Напишите страницу 3.11.3.html со сценарием, являющимся модификацией сценария из страницы 3.11.2.html. Исправьте в нем функцию `average` таким образом, чтобы она в качестве параметров могла принимать массивы с числами. Числовые элементы полученных массивов должны использоваться в вычислении среднего значения, нечисловые — игнорироваться. Чтобы получить размер массива, воспользуйтесь конструкцией вида *<переменная, в которой хранится массив>*`.length`. Модифицированной функции дайте имя `average2`. Проверьте функцию на следующих наборах параметров:

```
average2([10, 20], 30, [40, 50], 60, [70, 80, 90, 100])
average2('[10, 20]', 30, [40, '50'], 60, ['70, 80', 90, 100])
```

Урок 4
Классы и объекты

Объекты. Свойства и методы
Классы
Коллекции
Класс *Object*. Объектная нотация
Объектный тип данных
Значащие и ссылочные типы
Добавленные свойства

4.1. Объекты. Свойства и методы

В JavaScript любое значение: число, строка, логическая величина, функция или массив — является объектом (исключение представляет лишь значение `undefined`).

> *Объект* — сложная сущность, представляющая набор переменных, которые хранят различные характеристики этой сущности, и функций, которые выполняют над ней какие-либо действия.

Объявим переменную со строкой `'JavaScript'`:

```
> let str = 'JavaScript'
```

Эта строка представляет собой объект, поддерживающий ряд свойств и методов, которыми мы можем воспользоваться.

> *Свойство* — переменная, входящая в состав объекта и хранящая одну из его характеристик.

Для обращения к свойству применяется формат, аналогичный формату обращения к переменной (см. *разд. 1.7*):

```
<переменная с объектом>.<имя свойства>
```

Пользуясь им, мы можем как получать текущее значение свойства, так и присваивать ему новое значение.

> Точка, разделяющая *переменную с объектом* и *имя свойства*, — это *оператор доступа к элементу объекта*.

Например, свойство `length` объекта строки хранит ее длину, выраженную в символах:

```
> str.length
< 10
```

> *Метод* — функция, входящая в состав объекта и выполняющая определенное действие над ним.

Для вызова метода применяется формат, аналогичный формату вызова функции (см. *разд. 3.1*):

```
<переменная с объектом>.<имя метода>([<параметры метода через запятую>])
```

Так, метод `toLocaleUpperCase` строки возвращает копию объекта строки, у которого он был вызван (*текущего*) объекта, с символами, приведенными к верхнему регистру:

```
> str.toLocaleUpperCase()
< "JAVASCRIPT"
```

А метод `indexOf` возвращает позицию указанного символа в текущей строке.

Давайте выясним, на какой позиции находится буква `'S'`:

```
> str.indexOf('S')
< 4  // Символ №5 (нумерация символов начинается с 0)
```

Практически вся функциональность по обработке данных в JavaScript реализована в виде свойств и методов объектов.

4.1.1. Доступ к свойствам объекта посредством оператора [] (квадратные скобки)

Обращаться к свойствам объекта можно не только посредством оператора-точки, но и с помощью оператора доступа к элементу массива `[]` (см. *разд. 3.10*), записав между квадратными скобками строку с именем нужного свойства:

```
> let s = 'JavaScript'
> // Обращаемся к свойству length строки s
> s['length']
< 10
```

Имя свойства также может храниться в какой-либо переменной и даже формироваться по ходу выполнения сценария.

4.2. Классы

> *Класс* — образец для создания объектов определенного вида. Он задает набор свойств и методов, которые будут поддерживаться всеми объектами, относящимися к этому виду.

Класс должен иметь уникальное *имя*, которое обязано удовлетворять тем же требованиям, что и имя переменной (см. *разд. 1.7*).

Список базовых классов, встроенных в JavaScript и реализующих значения различных видов, приведен в табл. 4.1.

Таблица 4.1. Список базовых классов, встроенных в JavaScript

Имя класса	Вид значений	Имя класса	Вид значений
`Number`	Число	`String`	Строка
`Boolean`	Логическая величина	`Function`	Функция
`Array`	Массив	`Date`	Временна́я отметка

Например, строка `'JavaScript'` — это объект, созданный на основе класса `String`, или объект класса `String`.

Помимо этого, поддерживаются весьма специфические классы: `Math`, служащий для выполнения математических операций, и `Arguments`, чем-то похожий на массив.

Все эти классы, за исключением `Boolean` и `Function`, не поддерживающих никаких полезных свойств и методов, будут рассмотрены далее в этом уроке.

4.3. Класс *Number*

Класс `Number` представляет числа (значения числового типа).

Он поддерживает следующие полезные нам методы:

- `isInteger(<`*число*`>)` — статический, возвращает `true`, если заданное *число* целое, и `false` в противном случае.

  > *Статический метод* — вызывается не у объекта какого-либо класса, а у самого класса. Выполняет манипуляции лишь над переданными ему параметрами.

  Примеры:

```
> Number.isInteger(123)
< true
> Number.isInteger(1.23)
< false
```

- `toString([<`*основание*`>])` — преобразует текущее число в строку, которую и возвращает в качестве результата. Если задано *основание*, результат возвращается в системе счисления, соответствующей этому основанию. Если *основание* не задано, число возвращается в десятичной системе:

```
> let a = 123
> a.toString()
< "123"
> a.toString(8)
< "173"
> a.toString(16)
< "7b"
```

- `toExponential([<количество цифр после точки>])` — преобразует текущее число в экспоненциальный вид и возвращает в виде строки. Если *количество цифр после десятичной точки* не указано, веб-обозреватель выведет столько цифр, сколько нужно для представления этого числа:

```
> a = 123456.789123
> a.toExponential()
< "1.23456789123e+5"

> a.toExponential(3)
< "1.235e+5"
```

- `toFixed([<количество цифр после десятичной точки>])` — округляет текущее число до заданного *количества цифр после десятичной точки* (если не указано, округляет нацело) и возвращает в виде строки:

```
> a.toFixed()
< "123457"

> a.toFixed(2)
< "123456.79"
```

- `toPrecision([<длина числа в цифрах>])` — урезает текущее число до заданной *длины* и возвращает в виде строки. Если *длина* не указана, возвращает строку с неурезанным числом:

```
< a.toPrecision(7)
> "123456.8"

< a.toPrecision()
> "123456.789123"
```

Также могут оказаться полезными два статических свойства, поддерживаемые этим классом:

- `MIN_VALUE` — возвращает наименьшее число, которое способен обработать JavaScript;
- `MAX_VALUE` — возвращает наибольшее число, которое способен обработать JavaScript.

> *Статическое свойство* — принадлежит не объекту, а классу. Хранит значение, которое характеризуют сам класс.

Примеры:

```
> Number.MIN_VALUE
< 5e-324

> Number.MAX_VALUE
< 1.7976931348623157e+308
```

4.4. Класс *String*

Класс `String` представляет значения строкового типа, относящегося к значащим типам.

Свойство `length` этого класса возвращает длину строки в символах:

```
> 'JavaScript'.length
< 10
```

Класс `String` поддерживает весьма много методов:

- `toLocaleLowerCase()` — возвращает копию текущей строки, в которой все символы приведены к нижнему регистру с учетом языковых настроек системы:

  ```
  > 'JavaScript'.toLocaleLowerCase()
  < "javascript"
  ```

- `toLowerCase()` — то же самое, что и `toLocaleLowerCase`, но не учитывает языковые настройки. Пригоден только для преобразования строк, набранных латиницей;

- `toLocaleUpperCase()` — возвращает копию текущей строки, в которой все символы приведены к верхнему регистру с учетом языковых настроек системы:

  ```
  > 'JavaScript'.toLocaleUpperCase()
  < "JAVASCRIPT"
  ```

- `toUpperCase()` — то же самое, что и `toLocaleUpperCase`, но не учитывает языковые настройки. Пригоден только для преобразования строк, набранных латиницей;

- `trim()` — возвращает копию текущей строки, у которой пробелы в начале и конце удалены:

  ```
  > '     HTML и CSS     '.trim()
  < "HTML и CSS"
  ```

- `includes(<подстрока>)` — возвращает `true`, если текущая строка содержит указанную *подстроку*, и `false` в противном случае:

  ```
  > 'HTML и CSS'.includes('CSS')
  < true
  ```

  ```
  > 'HTML и CSS'.includes('Python')
  < false
  ```

- `startsWith(<подстрока>)` — возвращает `true`, если текущая строка начинается с указанной *подстроки*, и `false` в противном случае:

  ```
  > 'HTML и CSS'.startsWith('PHP')
  < false
  ```

- `endsWith(<подстрока>)` — возвращает `true`, если текущая строка оканчивается указанной *подстрокой*, и `false` в противном случае:

```
> 'HTML и CSS'.endsWith('CSS')
< true
```

- `indexOf(<подстрока>[, <начальная позиция для поиска>])` — возвращает номер позиции, на которой в текущей строке находится заданная *подстрока*. Поиск начинается с указанной *начальной позиции*, а если она не задана, — с начала строки (с символа с номером `0`). Если *подстрока* не найдена, возвращается `-1`:

```
> 'JavaScript'.indexOf('a')
< 1
> 'JavaScript'.indexOf('a', 2)
< 3
> 'JavaScript'.indexOf('z')
< -1
```

- `lastIndexOf(<подстрока>[, <начальная позиция для поиска>])` — то же самое, что `indexOf`, но поиск начинается с конца строки:

```
> 'JavaScript'.lastIndexOf('a')
< 3
```

- `charAt(<номер символа>)` — возвращает символ текущей строки, имеющий указанный *номер*. Нумерация символов начинается с `0`:

```
> 'HTML и CSS'.charAt(2)
< "M"
```

Вместо этого метода можно использовать оператор доступа к элементу массива `[]`, описанный в *разд. 3.10*:

```
> 'HTML и CSS'[2]
< "M"
```

- `substr(<номер первого символа фрагмента>[, <количество символов во фрагменте>])` — возвращает фрагмент текущей строки. Если *количество символов* не указано, возвращаемый фрагмент будет содержать остаток строки. Если *номер первого символа* отрицательный, он будет отсчитан от конца строки. Нумерация символов начинается с `0`:

```
> 'JavaScript'.substr(4)
< "Script"
> 'JavaScript'.substr(4, 3)
< "Scr"
> 'JavaScript'.substr(-4)
< "ript"
> 'JavaScript'.substr(-4,3)
< "rip"
```

- `substring(<номер первого символа фрагмента>[, <номер последнего символа фрагмента>])` — возвращает фрагмент текущей строки. Если *номер последнего*

символа не указан, возвращаемый фрагмент будет содержать остаток строки. Нумерация символов начинается с 0:

```
> 'JavaScript'.substring(4)
< "Script"

> 'JavaScript'.substring(4, 7)
< "Scr"
```

- replace(<*заменяемая подстрока*>, <*заменяющая подстрока*>) — заменяет в текущей строке первое вхождение *заменяемой подстроки* на *заменяющую* и возвращает результат:

```
> 'HTML и Python'.replace('Python', 'CSS')
< "HTML и CSS"
```

Чтобы заменить все вхождения *заменяемой подстроки*, следует использовать регулярное выражение (их применение будет описано в *уроке 10*);

- split(<*разделитель*>[, <*предельный размер массива*>]) — разбивает текущую строку на подстроки по заданному символу-*разделителю*, формирует из подстрок массив и возвращает его в качестве результата. Можно указать *предельный размер массива* — тогда избыточные подстроки в него не попадут:

```
> 'HTML,CSS,JavaScript'.split(',')
< (3) ["HTML", "CSS", "JavaScript"]

> 'HTML,CSS,JavaScript'.split(',', 2)
< (2) ["HTML", "CSS"]
```

- charCodeAt(<*номер символа*>) — возвращает числовой Unicode-код символа с указанным *номером*. Нумерация символов начинается с 0:

```
> 'HTML и CSS'.charCodeAt(5)
< 1080  // Код буквы «и»
```

- fromCharCode(<*код 1*>, <*код 2*> . . . <*код n*>) — статический, возвращает строку, составленную из символов с заданными Unicode-*кодами*:

```
> String.fromCharCode(0x42f, 0x437, 0x44b, 0x43a, 0x20,
      0x43, 0x53, 0x53)
< "Язык CSS"
```

4.5. Класс *Array*

Класс Array служит для представления массивов.

Свойство length этого класса возвращает размер массива, т. е. количество элементов в нем:

```
> let arr1 = [1, 2, 3, 4]
> arr1.length
< 4
```

Методов класс `Array` поддерживает много:

- `isArray(<объект>)` — статический, возвращает `true`, если *объект* является массивом, и `false` в противном случае:

```
> Array.isArray(arr1)
< true

> Array.isArray(1234)
< false
```

- `push(<значение 1>, <значение 2> . . . <значение n>)` — добавляет в конец массива указанные *значения* и возвращает новый размер массива:

```
> let arr2 = [10, 11]
> arr2.push(12, 13)
< 4

> arr2
< (4) [10, 11, 12, 13]
```

- `unshift(<значение 1>, <значение 2> . . . <значение n>)` — добавляет в начало массива указанные *значения* и возвращает новый размер массива:

```
> arr2.unshift(8, 9)
< 4

> arr2
< (6) [8, 9, 10, 11, 12, 13]
```

- `pop()` — удаляет последний элемент массива и возвращает его в качестве результата:

```
> arr2.pop()
< 13

> arr4
< (5) [8, 9, 10, 11, 12]
```

- `shift()` — удаляет первый элемент массива и возвращает его в качестве результата:

```
> arr4.shift()
< 8

> arr4
< (4) [9, 10, 11, 12]
```

- `splice(<индекс первого удаляемого элемента>[, <количество удаляемых элементов>[, <элемент 1>, <элемент 2> . . . <элемент n>]])` — удаляет из текущего массива заданное *количество* элементов и вставляет на их место заданные *элементы*. Если *индекс первого удаляемого элемента* отрицательный, он отсчитывается с конца массива. Если *количество удаляемых элементов* не указано, будет удален только элемент с заданным *индексом*, если оно равно `0` — никакие элементы удалены не будут. В качестве результата возвращается массив с удаленными элементами:

```
> let arr3 = ['HTML', 'C++', 'Python']
> arr3.splice(1, 1, 'CSS')
< ["C++"]

> arr3
< (3) ["HTML", "CSS", "Python"]

> arr3.splice(2, 0, 'JavaScript', 'PHP')
< []

> arr3
< (5) ["HTML", "CSS", "JavaScript", "PHP", "Python"]
```

- `includes(<значение>[, <начальный индекс>])` — возвращает `true`, если массив содержит элемент с заданным *значением*, и `false` в противном случае. Поиск ведется, начиная с элемента с указанным *начальным индексом*, если же таковой не указан — с самого первого элемента массива:

```
> let arr4 = [1, 1, 2, 3, 3, 3, 4]
> arr4.includes(1)
< true

> arr4.includes(1, 3)
< false
```

Метод `includes` для сравнения указанного *значения* с элементами текущего массива использует оператор `===` (строго равно). Следовательно, типы *значения* и элементов массива должны совпадать, в противном случае поиск не увенчается успехом:

```
> arr4.includes('3')
< false
```

- `indexOf(<значение>[, <начальный индекс>])` — возвращает индекс элемента текущего массива, хранящего заданное *значение*. Поиск начинается с элемента с указанным *начальным индексом*. Если *начальный индекс* отрицательный, он отсчитывается от конца массива. Если же он не указан, поиск начинается с первого элемента. Если подходящий элемент не найден, возвращается `-1`:

```
> arr4.indexOf(3)
< 3

> arr4.indexOf(3, 4)
< 4

> arr4.indexOf(3, -2)
< 5
```

Этот метод для сравнения указанного *значения* с элементами текущего массива также использует оператор `===` (строго равно);

- `lastIndexOf(<значение>[, <начальный индекс>])` — то же самое, что и `indexOf`, только поиск начинается с конца массива:

```
> arr4.lastIndexOf(3)
< 5
```

- every(<*функция проверки*>[, <*значение this*>]) — возвращает true, если все элементы текущего массива проходят проверку. Если хоть один элемент массива не проходит проверку, возвращается false.

  Проверку осуществляет указанная *функция*, принимающая три параметра: очередной элемент массива, индекс этого элемента и сам массив (два последних параметра необязательны). Функция должна возвращать логическое значение: true (если элемент прошел проверку) или false (в противном случае):

```
> // Проверяем, все ли элементы массива arr1 больше 0
> arr1.every((el, ind, ar) => el > 0)
< true

> // Проверяем, все ли элементы массива arr1 больше 2
> arr1.every((el) => el > 2)
< false
```

  Вторым параметром можно передать *значение*, которое будет доступно в теле *функции проверки* через локальную переменную this;

- some(<*функция проверки*>[, <*значение this*>]) — то же самое, что и every, только возвращает true, если проверку проходит хотя бы один элемент текущего массива:

```
> arr1.some((el, ind, ar) => el > 2)
< true
```

- find(<*функция проверки*>[, <*значение this*>]) — возвращает первый элемент текущего массива, который прошел проверку. Если ни один элемент не прошел проверку, возвращается undefined. Параметры этого метода те же, что и у метода every:

```
> arr1.find((el) => el > 3)
< 4

> arr1.find((el) => el > 5)
< undefined
```

- findIndex(<*функция проверки*>[, <*значение this*>]) — то же самое, что и find, но возвращает индекс первого прошедшего проверку элемента;

- filter(<*функция проверки*>[, <*значение this*>]) — возвращает массив, составленный из элементов текущего массива, которые прошли проверку. Параметры этого метода те же, что и у метода every:

```
> arr1.filter((el) => el > 2)
< (2) [3, 4]
```

- forEach(<*функция*>[, <*значение this*>]) — просто вызывает указанную *функцию* для каждого элемента текущего массива. Результата не возвращает. *Функция* должна принимать те же параметры, что и аналогичная функция у метода every, но не должна возвращать результат:

```
// Выводим на экран все элементы массива arr1
arr1.forEach((el) => {
    window.document.write('<p>', el, '</p>');
});
```

- `map(<функция>[, <значение this>])` — вызывает указанную *функцию* для каждого элемента текущего массива, помещает возвращенные ей результаты в новый массив и возвращает его. *Функция* должна принимать те же параметры, что и аналогичная функция у метода `every`, и возвращать результат действий с очередным элементом массива:

```
> // Создаем массив квадратов чисел из массива arr1
> arr1.map((el) => el ** 2)
< (4) [1, 4, 9, 16]
```

- `reduce(<функция>[, <начальное значение>])` — вызывает указанную *функцию* для каждого элемента текущего массива. *Функция* должна принимать следующие параметры:
  - результат, возвращенный предыдущим вызовом этой функции. Если это первый вызов (обрабатывается первый элемент массива), хранит заданное *начальное значение*, если же оно не указано — значение первого элемента массива;
  - значение очередного элемента текущего массива;
  - индекс очередного элемента текущего массива (необязательный);
  - ссылка на текущий массив (необязательный).

  Функция должна возвращать какой-либо результат, который будет передан ей при следующем вызове.

  В качестве результата метод `reduce` возвращает значение, возвращенное последним вызовом *функции*.

  Элементы текущего массива перебираются в направлении от начала к концу.

  Примеры:

```
> // Подсчитываем сумму элементов массива
> arr1.reduce((sum, el) => sum + el)
< 10
```

```
> // Создаем строку из пронумерованных элементов
> // массива, перечисленных через запятую
> arr3.reduce((str, el, i) =>  str + ++i + ': ' + el + ', ', '')
< "1: HTML, 2: CSS, 3: JavaScript, 4: PHP, 5: Python, "
```

- `reduceRight(<функция>[, <начальное значение>])` — то же самое, что и `reduce`, но элементы текущего массива перебираются от конца к началу:

```
> arr3.reduceRight((str, el, i) =>  str + ++i + ': ' + el + ', ', '')
< "5: Python, 4: PHP, 3: JavaScript, 2: CSS, 1: HTML, "
```

- `fill(<значение>[, <начальный индекс>[, <конечный индекс>]])` — заполняет элементы текущего массива между *начальным* и *конечным индексом* заданным *значением*, при этом элемент с *конечным индексом* не заполняется. Если *конечный индекс* не указан, заполнение ведется до конца массива. Если *начальный индекс* не указан, заполнение ведется начиная с первого элемента массива. Результата метод не возвращает:

```
> [10, 11, 12, 13, 14].fill(0)
< (5) [0, 0, 0, 0, 0]
> [10, 11, 12, 13, 14].fill(0, 2)
< (5) [10, 11, 0, 0, 0]
> [10, 11, 12, 13, 14].fill(0, 2, 4)
< (5) [10, 11, 0, 0, 14]
```

- `join([<разделитель>])` — объединяет все элементы текущего массива в строку, разделяя из заданным символом-*разделителем*, и возвращает эту строку в качестве результата. Если *разделитель* не задан, используется запятая:

```
> arr2.join()
< "HTML,CSS,JavaScript"
> arr2.join(', ')
< "HTML, CSS, JavaScript"
```

- `concat(<массив 1>, <массив 2> . . . <массив n>)` — создает новый массив, содержащий все элементы из текущего и всех переданных ему *массивов*, и возвращает его в качестве результата:

```
> arr1.concat([5, 6], [10, 11, 12])
< (9) [1, 2, 3, 4, 5, 6, 10, 11, 12]
```

- `slice([<начальный индекс>[, <конечный индекс>]])` — формирует из элементов текущего массива, расположенных между *начальным* и *конечным индексом*, новый массив и возвращает его. Если *конечный индекс* не указан, в новый массив помещаются все оставшиеся элементы. Если и *начальный индекс* не указан, создается копия текущего массива:

```
> arr1.slice(1, 3)
< (2) [2, 3]
> arr1.slice(1)
< (3) [2, 3, 4]
> arr1.slice()
< (4) [1, 2, 3, 4]
```

4.5.1. Сортировка массивов

Для сортировки массива применяется метод `sort` класса `Array`:

`sort([<функция сравнения>])`

В качестве результата метод возвращает текущий массив, уже отсортированный.

Если *функция сравнения* не указана, метод `sort` сортирует числа по возрастанию:

```
> let arr1 = [32, 7, 90, -4, 15, 900, 64]
> arr1.sort()
< (7) [-4, 15, 32, 64, 7, 90, 900]
```

Строки сортируются по возрастанию сумм кодов содержащихся в них символов — в результате выполняется сортировка в алфавитном порядке:

```
> let arr2 = ['HTML', 'Ruby', 'CSS', 'PHP', 'Python', 'C++']
> arr2.sort()
< (6) ["C++", "CSS", "HTML", "PHP", "Python", "Ruby"]
```

Если нужно отсортировать массив в другом порядке, в вызове метода `sort` следует задать *функцию сравнения*. Она должна принимать два параметра: два сравниваемых элемента массива — и возвращать:

- любое отрицательное число — если первый *элемент* меньше второго;
- `0` — если *элементы* равны;
- любое положительное число — если первый *элемент* больше второго.

Вот пример сортировки массива с числами по убыванию:

```
> arr.sort((el1, el2) => el2 - el1)
< (7) [900, 90, 64, 32, 15, 7, -4]
```

Теперь разберемся со строками. Выяснить, какая из них «меньше», а какая «больше», мы можем посредством обычных операторов сравнения наподобие < и > (подробнее об этом рассказано в *разд. 1.5*):

```
> 'CSS' < 'PHP'
< true
> 'Python' > 'Ruby'
< false
```

Но в *функции сравнения* для этой цели удобнее применять метод `localeCompare` класса `String`. Вот формат его вызова:

```
localeCompare(<сравниваемая строка>)
```

Метод возвращает:

- `-1` — если текущая строка «меньше» *сравниваемой*;
- `0` — если обе строки равны;
- `1` — если текущая строка «больше» *сравниваемой*.

Вот пример сортировки массива со строками в порядке, обратном алфавитному:

```
> arr2.sort((el1, el2) => el2.localeCompare(el1))
< (6) ["Ruby", "Python", "PHP", "HTML", "CSS", "C++"]
```

Метод `reverse` класса `Array` меняет порядок следования элементов массива на противоположный. Метод не принимает параметров и возвращает ссылку на текущий массив:

```
> arr2.reverse()
< (6) ["C++", "CSS", "HTML", "PHP", "Python", "Ruby"]
```

4.6. Класс *Date*

Класс `Date` служит для представления *временны́х отметок* (даты и времени наступления каких-либо событий).

Создать объект этого класса можно только с помощью оператора `new`.

> *Оператор создания объекта* `new` создает объект класса с указанным *именем* и возвращает его в качестве результата. Он записывается в формате:
> `new <имя класса>([<параметры объекта через запятую>])`

Параметры объекта, если они указаны, будут занесены в его свойства непосредственно при создании.

Выражение, создающее объект временно́й отметки, можно записать в трех форматах:

- `new Date()` — созданная временна́я отметка будет хранить текущие дату и время:

```
> let date1 = new Date();
```

- `new Date(<год>, <номер месяца, начиная с 0>, <число>[, <часы>, <минуты>, <секунды>[, <миллисекунды>]])` — временна́я отметка будет хранить указанные дату и время:

```
> // 1 января 2019 года, 11:45:09.400
> let date2 = new Date(2019, 0, 1, 11, 45, 9, 400);
```

Если не указать *миллисекунды*, количество миллисекунд во временно́й отметке будет установлено равным `0` (впрочем, их указывают крайне редко):

```
> // 1 января 2019 года, 11:45
> let date3 = new Date(2019, 0, 1, 11, 45, 0);
```

Если также не указать *часы*, *минуты* и *секунды*, временна́я отметка будет хранить полночь указанной даты:

```
> // 28 февраля 2019 года, полночь
> let date4 = new Date(2019, 1, 28);
```

- `new Date(<строка с датой и временем>])` — *строка с датой и временем* указывается в формате:

<год>-<№ месяца>-<число>[<часы>:<минуты>:<секунды>]

Месяцы в этом случае нумеруются, *начиная с 1*:

```
> // 12 марта 2019 года, 12:05:40
> let date5 = new Date('2019-03-12 12:05:40');

> // 12 марта 2019 года, полночь
> let date6 = new Date('2019-03-12');
```

Методы класса `Date`, предназначенные для получения или изменения одной из составляющих временно́й отметки: года, месяца, часов и др. приведены в табл. 4.2.

***Таблица 4.2.** Методы класса `Date`*

Составляющая	Получение	Изменение
Год	`getFullYear()`	`setFullYear(<год>)`
№ месяца	`getMonth()`	`setMonth(<№ месяца>*)`
Число	`getDate()`	`setDate(<число>)`
Часы	`getHours()`	`setHours(<часы>)`
Минуты	`getMinutes()`	`setMinutes(<минуты>)`
Секунды	`getSeconds()`	`setSeconds(<секунды>)`
Миллисекунды	`getMilliseconds()`	`setMilliseconds(<мсек>)`

*** Нумерация месяцев начинается с нуля: 0 — январь, 1 — февраль и т. д.**

Примеры:

```
> // Какой сейчас год?
> date1.getFullYear()
< 2019

> // А месяц?
> date1.getMonth()
< 2  // Третий по счету месяц – март

> // Меняем месяц на июль, а число – на 24-е
> date2.setMonth(6)
> date2.setDate(24)
```

Еще несколько полезных методов:

- `getDay()` — возвращает порядковый номер дня недели от `0` до `6`: `0` — воскресенье, `1` — понедельник и т. д.;
- `toLocaleString()` — преобразует текущую временну́ю отметку в строку согласно языковым параметрам системы:

```
> date1.toLocaleString()
< "19.03.2019, 11:14:45"
```

- `toLocaleDateString()` — преобразует дату из текущей временно́й отметки в строку согласно языковым параметрам системы:

```
> date1.toLocaleDateString()
< "19.03.2019"
```

- `toLocaleTimeString()` — преобразует время из текущей временно́й отметки в строку согласно языковым параметрам системы:

```
> date1.toLocaleTimeString()
< "11:14:45"
```

4.7. Класс *Math*

Класс `Math` содержит только статические свойства, хранящие математические константы, и статические методы, реализующие действия математики и тригонометрии.

Статические свойства класса `Math`:

- `PI` — число π:

```
> Math.PI
< 3.141592653589793
```

- `E` — число e (основание натурального логарифма);
- `SQRT2` — квадратный корень от 2;
- `SQRT1_2` — квадратный корень от 0,5;
- `LN2` — ln 2 (натуральный логарифм от 2);
- `LN10` — ln 10 (натуральный логарифм от 10);
- `LOG2E` — lb e (двоичный логарифм от числа e);
- `LOG2E` — lg e (десятичный логарифм от числа e).

Методы класса `Math` и возвращаемые ими результаты:

- `sqrt(<число>)` — квадратный корень от *числа*:

```
> Math.sqrt(20)
< 4.47213595499958
```

- `cbrt(<число>)` — кубический корень от *числа*;
- `min(<произвольное количество чисел>)` — наименьшее *число* из указанных:

```
> Math.min(2, 56, 900, 425, -87)
< -87
```

- `max(<произвольное количество чисел>)` — наибольшее *число* из указанных;
- `abs(<число>)` — модуль (абсолютная величина) *числа*;
- `round(<число>)` — *число*, округленное до ближайшего целого;
- `floor(<число>)` — *число*, округленное до ближайшего меньшего целого;
- `ceil(<число>)` — *число*, округленное до ближайшего большего целого;
- `trunc(<число>)` — целая часть *числа*;
- `sin(<угол>)` — синус *угла*, заданного в радианах.

Пояснение

Пересчитать угол из градусов в радианы можно по формуле: $\alpha_{гр} \times \pi / 180$.

Вот пример вычисления синуса угла 45°:

```
> let angle = 45 * Math.PI / 180
> angle
< 0.7853981633974483

> Math.sin(angle)
< 0.7071067811865475
```

- `cos(<угол>)` — косинус *угла*, заданного в радианах;
- `tan(<угол>)` — тангенс *угла*, заданного в радианах.
- `asin(<число>)` — арксинус в радианах от *числа*;
- `acos(<число>)*` — арккосинус в радианах от *числа*;
- `atan(<число>)` — арктангенс в радианах от *числа*.

Пояснение

Пересчитать угол из радианов в градусы можно по формуле: $\alpha_{рад} \times 180 / \pi$.

Вот пример вычисления арктангенса от числа 1 и получения результата в градусах:

```
> let at = Math.atan(1)
> at * 180 / Math.PI
< 45
```

- `sinh(<число>)` — гиперболический синус от *числа*;
- `cosh(<число>)` — гиперболический косинус от *числа*;
- `tanh(<число>)` — гиперболический тангенс от *числа*;
- `asinh(<число>)` — гиперболический арксинус от *числа*;
- `acosh(<число>)` — гиперболический арккосинус от *числа*;
- `atanh(<число>)` — гиперболический арктангенс от *числа*;
- `atan2(<y>, <x>)` — угол в радианах между осью абсцисс и точкой с координатами [*x*, *y*], отсчитанный против часовой стрелки;
- `random()` — псевдослучайное дробное число от 0 до 1:

```
> Math.random()
< 0.2220291778805139

> Math.random()
< 0.5721614375670632

> Math.random()
< 0.19212452555545023
```

- `log(<число>)` — натуральный логарифм *числа*;
- `exp(<число>)` — значение $e^{число}$;
- `pow(<a>, <b>)` — значение a^b. Метод остался от старых версий JavaScript и может встретиться в старом коде.

4.8. Класс *Arguments*. Коллекции

Класс `Arguments` представляет список значений параметров, полученных функцией при вызове.

Объект этого класса создается самим веб-обозревателем при исполнении тела функции и присваивается локальной переменной `arguments` (из *разд. 3.2* мы знаем, что локальные переменные существуют только внутри тела функции).

Класс `Arguments` поддерживает свойство `length`, возвращающее количество параметров, полученных функцией.

Для получения собственно значений параметров применяется оператор доступа к элементу `[]`, знакомый нам по *разд. 3.10*.

Вот пример функции, использующей объект класса `Arguments` для доступа к переданным ей параметрам:

```
function manyParameters() {
    let l = arguments.length, i, el;
    for (i = 0; i < l; i++)
        el = arguments[i];
        . . .
}
```

Как видим, объект этого класса чем-то похож на обычный массив.

> *Коллекция* — объект, имеющий функциональность массива.
> Все классы коллекций поддерживают оператор `[]` и свойство `length`.

4.9. Класс *Object*. Служебные объекты. Объектная нотация

Класс `Object` — самый простой из всех поддерживаемых JavaScript классов и одновременно наиболее фундаментальный. Все остальные классы: `String`, `Date`, `Array` и др. — являются его наследниками.

Объекты класса `Object` активно применяются для хранения набора каких-либо значений, представляющих определенную сущность. Такие объекты носят название *служебных*.

> Создать служебный объект класса `Object` проще всего, применив `объектную нотацию`, которая записывается в формате:
>
> ```
> {
>  <имя свойства 1>: <значение свойства 1>,
>  <имя свойства 2>: <значение свойства 2>,
>  . . .
>  <имя свойства n-1>: <значение свойства n-1>,
>  <имя свойства n>: <значение свойства n>
> }
> ```

Пары *<имя свойства>*: *<значение свойства>* отделяются друг от друга запятыми, а не точками с запятой. После последней пары запятая не ставится.

Пример:

- создаем служебный объект, хранящий сведения о каком-либо человеке:

```
let person = {
    name1: 'Иван',
    name2: 'Иванов',
    age: 48
};
```

- обращаемся к свойству `name2` этого объекта:

```
let n = person.name2;
```

- и передаем объект функции в качестве параметра:

```
let result = computePerson(person);
```

Служебные объекты применяются в веб-разработке весьма активно, т. к. могут хранить любой набор значений, а создать их очень просто.

`Object` — наиболее фундаментальный из всех классов JavaScript. Все остальные классы являются его наследниками.

Наследование — создание одного класса на основе другого. Унаследованный класс (*производный*, или *подкласс*), помимо свойств и методов, определенных в нем самом, получает все свойства и методы класса, от которого он наследован (*базового*, или *суперкласса*).

4.10. Объектный тип. Значение *null*

Все объекты, не являющиеся строкой, числом, логическим значением или функцией, относятся к *объектному типу*. Оператор `typeof` возвращает для таких типов строку `'object'`.

Пример:

```
> date1 = new Date()
> typeof date1
< "object"
> typeof [1, 2, 3]
< "object"
```

Значение `null`, также принадлежащее к объектному типу, обозначает *нулевую ссылку*, не указывающую ни на один объект.

Объекты могут использоваться в качестве условий в условных операторах и выражениях, а также в арифметических выражениях.

Любой объект преобразуется в логическое значение `true` и числовое значение `NaN`, в зависимости от типа выражения, в котором он поставлен.

Нулевая ссылка `null` преобразуется в `false` и `0`.

Примеры:

```
> (date1) ? 'Объект существует' : 'Объект не существует'
< "Объект существует"
> (null) ? 'Объект существует' : 'Объект не существует'
< "Объект не существует"
> 12 * null
< 0
```

4.11. Хранение объектов в переменных. Значащие и ссылочные типы

Объект может храниться в переменной по-разному, в зависимости от того, к какой разновидности принадлежит его тип: значащей или ссылочной.

> Объекты *значащих* типов хранятся *непосредственно* в переменных. При присваивании такого объекта другой переменной в последнюю помещается его полная копия.
>
> К значащим типам относятся числа, строки и логические значения.

Пример:

```
> let a = 1, b;
> b = a
> // Значение из переменной a копируется в переменную b.
> // Теперь в переменных a и b хранятся разные объекты.
> // Для проверки присвоим переменной a другое значение.
> a = 2
> // Значение в переменной a изменилось...
> a
< 2
> // ...при этом в переменной b осталось старое значение
> b
< 1
```

> Объекты *ссылочных* типов хранятся в отдельной области памяти, а в переменные записываются *ссылки* на них. При присваивании такого объекта ссылка из правой переменной копируется в левую, в результате чего обе переменные станут указывать на один и тот же объект.
>
> К ссылочным относятся функциональный и объектный типы.

Пример:

```
> date1 = new Date()
> date2 = date1
```

```
> // Переменные date1 и date2 должны хранить ссылку на один и тот же
> // объект временной отметки. Проверим это.
> // Выясним месяц посредством обращения к обеим переменным.
> date1.getMonth()
< 2  // Март
> date2.getMonth()
< 2
> // Теперь изменим месяц на 1 (февраль), обратившись к переменной date1
> date1.setMonth(1)
> // и выясним месяц, обратившись к переменной date2
> date2.getMonth()
< 1
> // Мы получили то же число 1 (февраль).
> // Следовательно, переменные date1 и date2 действительно
> // ссылаются на один и тот же объект.
```

> В дальнейшем ради простоты мы будем говорить, что какая-либо переменная хранит, а какая-либо функция возвращает объект, а не ссылку на него.

4.12. Добавленные свойства

В дополнение к свойствам, полученным от класса, мы можем создавать в объектах новые свойства.

> *Добавленное свойство* — произвольное свойство, созданное в объекте и расширяющее набор свойств, полученных от класса. Создается простым присваиванием значения.
>
> Добавленные свойства могут быть созданы только у объектов ссылочных типов (функционального и объектного).

Для примера добавим свойство `description` созданному ранее объекту `date1`, представляющему текущую временну́ю отметку:

```
> date1.description = 'Текущая дата'
```

После чего сможем обратиться к этому свойству в другом месте кода:

```
> date1.description
< "Текущая дата"
```

Свойство `description` присутствует только в объекте `date1`. Давайте убедимся в этом на примере другого объекта временно́й отметки:

```
> date2 = new Date(2019, 1, 27, 12, 0, 0)
> date2.description
< undefined  // У объекта date2 такого свойства нет
```

> *Оператор удаления свойства* `delete` удаляет из объекта указанное добавленное *свойство*:
>
> `delete <переменная с объектом>.<удаляемое свойство>`

Удаляем добавленное свойство `description` объекта `date1`:

```
> delete date1.description
```

Добавленные свойства мы можем создавать у любых объектов, в том числе служебных (класса `Object`):

```
// Добавляем в ранее созданный объект person свойство workplaces,
// хранящее массив с названиями мест работы
person.workplaces = [
    'Шараш-Монтаж',
    'Монтаж-Шараш',
    'Министерство свободного времени'
];
```

4.13. Дополнительные средства для работы с объектами

> *Оператор проверки существования свойства* `in` возвращает `true`, если свойство с указанным *именем* существует в *объекте*, и `false` в противном случае. Он записывается в формате:
>
> *<имя свойства>* `in` *<переменная с объектом>*
>
> *Имя свойства* здесь указывается в виде строки.

Проверяем, существует ли свойство `name1` в объекте `person`, созданном в *разд. 4.9*:

```
> 'name1' in person
< true
```

Проверяем, существует ли в том же объекте свойство `address`:

```
> 'address' in person
< false
```

> *Оператор принадлежности классу* `instanceof` возвращает `true`, если заданный *объект* принадлежит указанному *классу*, и `false` в противном случае. Он записывается в формате:
>
> *<переменная с объектом>* `instanceof` *<класс>*

Проверяем, принадлежит ли объект `person` классу `Object`:

```
> person instanceof Object
< true
```

Проверяем, принадлежит ли число `person` классу `Window`:

```
> person instanceof Window
< false
```

Цикл по свойствам последовательно перебирает все свойства, что присутствуют в *объекте*. Вот формат его записи:

```
for (<переменная> in <объект>)
  <тело цикла>
```

На каждом проходе цикла имя очередного свойства, представленное в виде строки, присваивается указанной *переменной*, которая доступна только в *теле цикла*.

Перебираем все свойства объекта `person` и выводим их на экран в формате *<имя>*: *<значение>*:

```
for (pr in person)
    window.document.write('<p>', pr, ': ', person[pr], '</p>');
```

4.14. Самостоятельные упражнения

- Возьмите страницу 2.1.html с веб-сценарием, преобразующим величины из дюймов в сантиметры. Доработайте его таким образом, чтобы в случае, если пользователь введет в дробном числе запятую вместо десятичной точки, он корректно преобразовывал бы такое число. Файл исходной страницы 2.1.html вы найдете в папке 4\!sources сопровождающего книгу электронного архива (см. *приложение 3*).

  Подсказка: вспомните, что строка является объектом класса `String`, и используйте метод, заменяющий одну подстроку на другую.

- Напишите страницу 4.14.1.html со сценарием, который получит от пользователя список чисел (посредством диалогового окна ввода значения, выводимого конструкцией `window.prompt`), разделенных запятыми, и выведет на страницу их среднее арифметическое. При этом не используйте в сценарии циклы.

  Подсказка: превратите объект полученной от пользователя строки в массив, содержащий отдельные числа, и подумайте, какой метод массива (класса `Array`) подойдет в этом случае.

- Напишите страницу 4.14.2.html со сценарием, который выведет текущую дату в формате:

  `Сегодня <число> <название месяца> <год> г.`

  Подсказка: создайте объект временно́й отметки и используйте методы класса `Date`.

- Напишите страницу 4.14.3.html со сценарием, который выведет в виде таблицы синусы, косинусы и тангенсы углов 0°, 15°, 30°, 45°, 60°, 75° и 90°.

- Напишите страницу 4.14.4.html со сценарием, который получит от пользователя список чисел, разделенных запятыми, и выведет их отсортированными по возрастанию, причем сначала выведет все четные числа, а потом — все нечетные.

  Подсказка: превратите полученную от пользователя строку в массив, содержащий отдельные числа, и отсортируйте его.

Урок 5
Средства отладки

Отладчик веб-обозревателя
Трассировка
Инспектор DOM
Исключения и их обработка

5.1. Вывод сообщений в консоли

В случае ошибки веб-обозреватель немедленно останавливает выполнение сценария и показывает в консоли соответствующее сообщение. Оно выводится красными символами на бледно-красном фоне и включает текстовое описание ошибки, имя файла и номер строки исходного кода, при исполнении которой произошла ошибка (рис. 5.1).

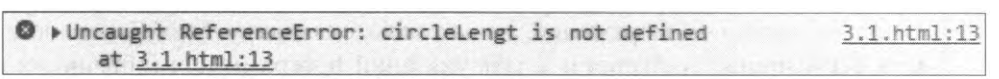


Рис. 5.1. Пример сообщения об ошибке

Как можно видеть, здесь зафиксирована попытка вызвать необъявленную функцию `circleLengt` в строке 13 файла 3.1.html.

Вообще, при разработке сценариев рекомендуется держать консоль открытой (как ее открыть, описывалось в *разд. 1.1*), чтобы сразу видеть все сообщения о возникающих ошибках.

5.1.1. Вывод в консоли произвольных сообщений

Мы можем вывести в консоли произвольное сообщение — например, величину, вычисляемую сценарием (чтобы посмотреть, правильно ли она вычисляется). Сообщение будет показано с указанием имени файла и номера строки кода, в которой был выполнен его вывод.

Для вывода сообщения в консоли применяется метод `log` объекта класса `Console`, представляющего консоль:

```
window.console.log(<выводимое сообщение>)
```

Выводимое сообщение может быть строкой, числом или логической величиной.

Объект консоли хранится в свойстве `console` объекта класса `Window`, находящегося в переменной `window` и представляющего текущее окно веб-обозревателя. Этот объект и эта переменная создаются самим веб-обозревателем.

Пример:

```
window.console.log(circleLength(2));
```

Более подробно класс `Window`, его свойства и методы мы рассмотрим в *уроке 11*.

5.2. Упражнение. Работаем с отладчиком веб-обозревателя

Возьмем страницу 3.1.html, написанную в *разд. 3.8*, выполним код содержащегося в ней сценария пошагово — выражение за выражением, и посмотрим, что при этом происходит. Для этого нам понадобится отладчик веб-обозревателя.

> *Отладчик* — позволяет приостанавливать исполнение сценария в указанных точках останова, трассировать его и просматривать значения, хранящиеся в переменных.

Отладчик, как и знакомая нам консоль, входит в состав отладочных инструментов веб-обозревателя.

> *Точка останова* — выражение, на котором отладчику следует приостановить выполнение сценария и ждать указаний программиста (обычно на выполнение трассировки).

> *Трассировка* — пошаговое, выражение за выражением, исполнение кода сценария.

1. В папке 5\!sources сопровождающего книгу электронного архива (см. *приложение 3*) найдем страницу 3.1.html и файл сценария 3.8.js, после чего откроем страницу в веб-обозревателе.
2. Находясь в веб-обозревателе, нажмем клавишу <F12>, чтобы вывести панель с отладочными инструментами.
3. Переключимся на вкладку **Sources**, где и располагается отладчик (рис. 5.2).

Рис. 5.2. Корешок вкладки **Sources**

Интерфейс отладчика включает полосу кнопок отладки и три панели, выводящие разные данные (рис. 5.3):

- *панель файлов* — показывает список всех файлов, хранящих код открытой страницы (в нашем случае файлы самой страницы и веб-сценария);

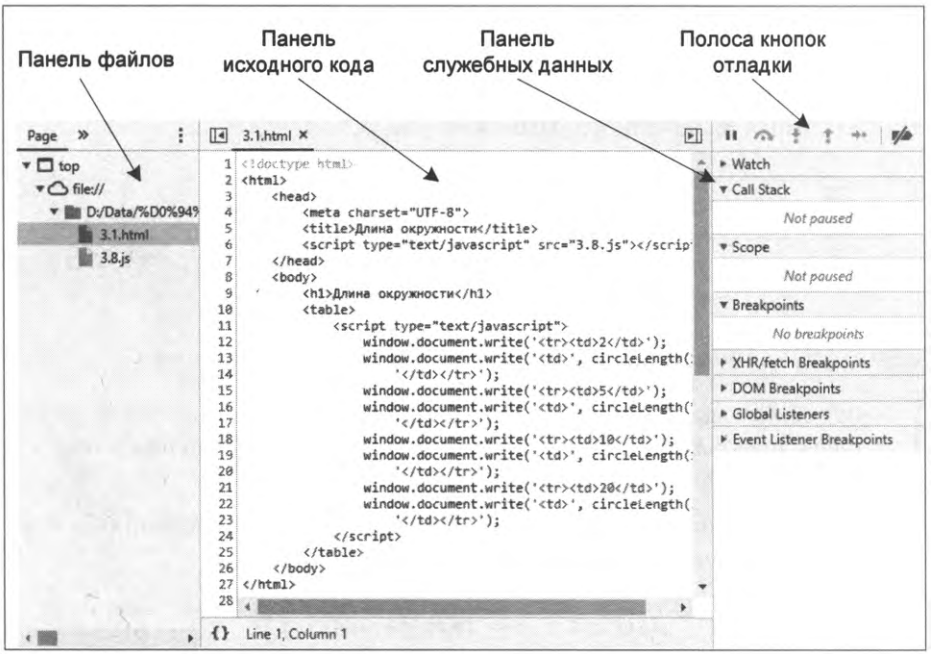


Рис. 5.3. Панели отладчика и полоса кнопок отладки

- *панель исходного кода* — показывает содержимое файла, выбранного в панели файлов;
- *панель служебных данных* — выводит всевозможную служебную информацию.

Мы можем менять размеры этих панелей относительно друг друга, перетаскивая мышью разделяющие их вертикальные линии.

Если панель исходного кода показывает не содержимое выбранного файла, а пустой серый фон, нужно в панели файлов выбрать любой другой пункт (файл или папку — это неважно), после чего вновь переключиться на файл, выбранный ранее.

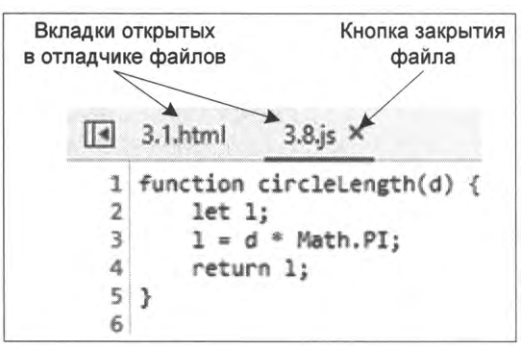


Рис. 5.4. Вкладки открытых в отладчике файлов

4. Как можно видеть на рис. 5.3, панель исходного кода выводит HTML-код страницы 3.1.html, которая выбрана в панели файлов. Также в панели файлов мы видим файл сценария 3.8.js. Давайте выведем и его.

Для этого щелкнем на файле 3.8.js в панели файлов — хранящийся в нем JavaScript-код появится на отдельной вкладке панели исходного кода (рис. 5.4).

5. Поскольку нам надо работать со сценарием, содержащемся в файле 3.1.html, переключимся вновь на этот файл, щелкнув на соответствующей вкладке.

   Пусть мы хотим начать трассировку сценария с самого первого его выражения — вот этого, выделенного рамкой на рис. 5.5.

```
11          <script type="text/javascript">
12              window.document.write('<tr><td>2</td>');
13              window.document.write('<td>', circleLength(
```

Рис. 5.5. Первое выражение сценария

Следовательно, именно на нем нужно поставить точку останова.

> *Точка останова* ставится щелчком мышью на расположенной слева белой полосе с номерами строк кода напротив нужного выражения и обозначается синей стрелкой, направленной вправо.

6. Щелкнем мышью на белой полосе напротив первого выражения — и там появится точка останова (рис. 5.6).

```
11          <script type="text/javascript">
12              window.document.write('<tr><td>2</td>');
13              window.document.write('<td>', circleLength(
```

Рис. 5.6. На первом выражении сценария поставлена точка останова

Одновременно в области **Breakpoints** панели служебных данных появится флажок, представляющий эту точку останова. Напротив флажка будет показано имя файла, номер строки кода, на которой стоит эта точка, и, ниже, фрагмент этого кода (рис. 5.7).

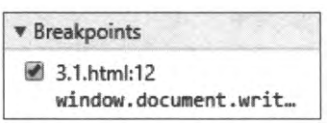


Рис. 5.7. Область **Breakpoints** панели служебных данных с флажком, представляющим точку останова

> Сбросив флажок точки останова в области **Breakpoints**, можно временно деактивировать эту точку, не удаляя ее.

7. Выполним наш веб-сценарий повторно, перезагрузив страницу.

   Выполнение сценария приостановится на точке останова. Строка с этим выражением в панели исходного кода будет выделена синей рамкой с голубым фоном (рис. 5.8).

```
11          <script type="text/javascript">
12              window.document.write('<tr><td>2</td>');
13              window.document.write('<td>', circleLength(
```

Рис. 5.8. Выполнение сценария приостановлено на точке останова

8. Выполним шаг с заходом.

> *Шаг с заходом* — выполнение одного выражения в процессе трассировки кода, при котором код функции, чей вызов присутствует в этом выражении, также будет трассироваться.

Выполняется шаг с заходом нажатием кнопки **Step into next function call** (рис. 5.9), которая находится на полосе кнопок отладки (см. рис. 5.3), или клавиши <F11>.

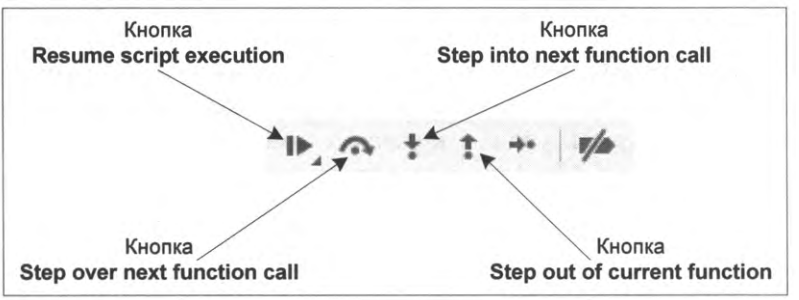


Рис. 5.9. Кнопки полосы кнопок отладки

Выделенное выражение будет исполнено, и выделение переместится на следующее (рис. 5.10). В нем находится вызов функции `circleLength`. Если мы снова выполним шаг с заходом, веб-обозреватель станет трассировать код этой функции.

```
11          <script type="text/javascript">
12              window.document.write('<tr><td>2</td>');
13              window.document.write('<td>', circleLength(
14                  '</td></tr>');
```

Рис. 5.10. Был выполнен шаг с заходом

9. Проверим, так ли это, снова выполнив шаг с заходом. Отладчик переключится на вкладку с файлом 3.8.js, в котором хранится объявление функции `circleLength`, и выделит первое выражение ее тела (рис. 5.11).

```
1  function circleLength(d) {
2      let l;
3      l = d * Math.PI;
4      return l;
5  }
```

Рис. 5.11. Шаг с заходом на первое выражение тела функции `circleLength`

10. Выполним шаг с заходом еще два раза, чтобы выделение переместилось на последнее выражение тела функции: `return l;`.

Веб-обозреватель покажет нам текущие значения параметра `d` и локальной переменной, объявленной в этой функции. Так мы сразу поймем, правильно ли выполняются вычисления (рис. 5.12).

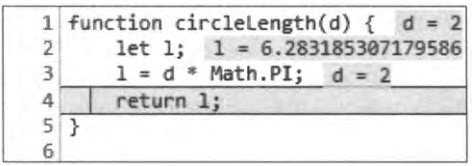


Рис. 5.12. Текущие значения параметра `d` и локальной переменной, объявленной в функции `circleLength`

Текущие значения переменных выводятся непосредственно в исходном коде на оранжевом фоне.

Посмотреть значение какой-либо переменной также можно, наведя курсор мыши на имя этой переменной, — значение будет выведено в небольшом окошке, которое можно закрыть уводом курсора с имени переменной или нажатием клавиши <Esc> (рис. 5.13). Таким же образом можно просматривать и значения свойств объектов.

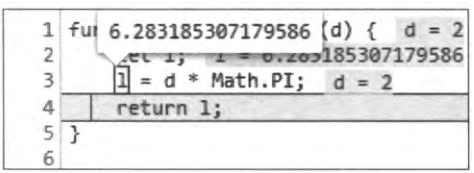


Рис. 5.13. Просмотр значения переменной наведением курсора мыши

Как можно видеть, функция вычисляет длину окружности правильно, и трассировать ее далее смысла нет.

11. Выполним выход из функции.

Выход из функции — выполнение оставшегося кода функции в обычном режиме и его приостановка на выражении, в котором был выполнен ее вызов.

Выход из функции производится нажатием кнопки **Step out of current function** на полосе кнопок отладки (см. рис. 5.9) или комбинации клавиш <Shift>+<F11>.

Выделение останется на втором по счету выражении, в котором вызывалась функция `circleLength`.

12. Поскольку нам более не нужно производить трассировку функции, в дальнейшем мы будем трассировать код, выполняя шаги с обходом.

Шаг с обходом — выполнение одного выражения в процессе трассировки кода, при котором код вызываемых функций не трассируется, а исполняется в обычном режиме.

Выполнение шага с обходом производится нажатием кнопки **Step over next function call** на полосе кнопок отладки (см. рис. 5.9) или клавиши <F10>.

Итак, выполним шаг с обходом трижды, пока выделение не остановится на пятом по счету выражении сценария из страницы 3.1.html.

Судя по всему, в остальном коде ошибок не будет, и исполнять его пошагово смысла нет.

13. Возобновим работу сценария.

> *Возобновление* — отмена трассировки и выполнение остального кода в обычном режиме.

Возобновление работы сценария производится нажатием кнопки **Resume script execution** на полосе кнопок отладки (см. рис. 5.9) или нажатием клавиши <F8>.

5.2.1. Удаление точек останова

Веб-обозреватель сохраняет все точки останова, поставленные в коде. При следующем открытии этой же страницы все точки останова будут восстановлены.

Удалить точку останова можно повторным щелчком на представляющей ее синей стрелке, расположенной на левой полосе с номерами строк.

Также можно щелкнуть правой кнопкой мыши на флажке, представляющем нужную точку останова в области **Breakpoints** панели служебных данных, и выбрать в появившемся контекстном меню пункт:

- **Remove breakpoint** — удаление этой точки останова;
- **Remove all breakpoints** — удаление всех точек останова;
- **Remove other breakpoints** — удаление всех точек останова, кроме этой.

В качестве практического упражнения удалите точку останова, поставленную при выполнении этого упражнения.

Также точку останова можно временно деактивировать и активировать вновь, сбрасывая и устанавливая представляющий ее флажок в области **Breakpoints** панели служебных данных.

5.3. Инспектор DOM

Многие веб-сценарии изменяют содержание страницы, добавляя, изменяя и удаляя ее элементы. Посмотреть на текущее содержание страницы можно с помощью инспектора DOM.

> *Инспектор DOM* — отладочный инструмент, показывающий в иерархическом виде составляющие страницу HTML-теги. Для его вызова нужно выбрать вкладку **Elements** панели отладочных инструментов (см. рис. 5.2).

Вот так выглядит инспектор DOM при открытии страницы 3.1.html (рис. 5.14).

HTML-код страницы показывается в виде иерархического «дерева». Мы можем разворачивать отдельные его «ветви», чтобы посмотреть содержимое соответствующих тегов.

Рис. 5.14. Инспектор DOM при открытии страницы 3.1.html

При наведении курсора мыши на какой-либо тег созданный им элемент выделяется голубым фоном, а на небольшой панели выводится имя создавшего его тега и геометрические размеры в пикселах.

Например, при наведении курсора мыши на тег `<table>` созданная им таблица с величинами длин окружностей будет выделена (рис. 5.15).

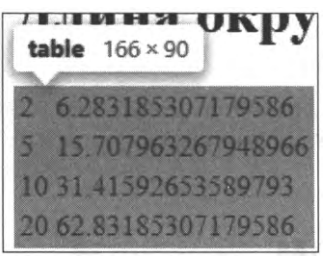


Рис. 5.15. Панель с выделенной таблицей, созданной тегом `<table>`

5.4. Исключения и их обработка

Обычно при возникновении ошибки выполнение веб-сценария останавливается, в результате чего может произойти потеря данных или даже искажение страницы. Избежать аварийной остановки сценария в этом случае можно, обрабатывая исключения.

> *Исключение* — объект, генерируемый веб-обозревателем в случае возникновения ошибки и хранящий сведения о ней.

Объект исключения принадлежит одному из классов, производных от класса `Error` (о наследовании классов говорилось в *разд. 4.9*). Каждый из классов исключений представляет отдельный вид ошибок.

> *Обработка исключения* — выполнение, в случае возникновения исключения, его *обработчика* — фрагмента кода, производящего какие-либо действия с целью устранить последствия ошибки (например, выводящего сообщение).

При наличии обработчика исключения исполнение сценария в случае ошибки не останавливается.

Обработчик исключения записывается в следующем формате:

```
try
    <блок try>
catch(<переменная для хранения исключения>)
    <блок catch>
[finally
    <блок finally>]
```

Если в *блоке try* возникнет исключение, будет выполнен *блок catch* — обработчик исключений, а потом — *блок finally* (если он указан). Если в *блоке try* исключения не возникнет, выполнится только *блок finally* (опять же, если он указан). В *блоке finally* записывается код, который должен выполниться независимо от того, возникла ли ошибка или нет.

Переменной, указанной в скобках после слова `catch`, будет присвоен объект исключения. Эта переменная будет доступна только в *блоке catch* и может быть использована для получения сведений об ошибке.

Все классы исключений поддерживают два свойства:

- `name` — хранит имя класса исключения в виде строки. Вот все поддерживаемые JavaScript классы исключений:
  - `SyntaxError` — в строке с выражением, переданной встроенной функции `eval` (см. *разд. 3.9*), присутствует ошибка:

    ```
    eval('1 # 1');
    // Здесь возникнет исключение SyntaxError,
    // поскольку оператор # в JavaScript отсутствует
    ```

  - `ReferenceError` — обращение к несуществующей переменной, функции, свойству, методу или классу. Возникает только в веб-сценариях;
  - `TypeError` — неподходящий тип значения:

    ```
    let n = 1;
    let s = n.toUpperCase();
    // Здесь возникнет исключение TypeError,
    // поскольку метод toUpperCase не поддерживается числами
    ```

- `RangeError` — для параметра функции или метода указано недопустимое значение:

```
let n = 1;
let s = n.toPrecision(500);
// Здесь возникнет исключение RangeError,
// поскольку в качестве параметра метода toPrecision задано
// слишком большое значение
```

- `URIError` — ошибка декодирования GET- или POST-параметра (о GET-, POST-параметрах и передаче данных разговор пойдет на *уроке 19*);
- `EvalError` — то же, что и `SyntaxError`. Поддерживалось более старыми версиями JavaScript и может встретиться в старом коде;

♦ `message` — описание возникшей ошибки в виде строки.

Вот пример кода, обрабатывающий ситуацию, когда функция `someFunc`, используемая в вычислениях, не объявлена:

```
let n;
window.document.write('<p>');
try {
    n = someFunc(123);
    window.document.write(n);
}
catch(exception) {
    if (exception.name == 'ReferenceError')
        window.document.write('Функция someFunc не объявлена. ',
            'Привяжите к странице файл somefunc.js с ее ',
            'объявлением.');
    else
        window.document.write('Возникла ошибка ', exception.name,
            ': ', exception.message);
}
finally {
    window.document.write('</p>');
}
```

Теперь, если мы забудем привязать к странице файл с объявлением функции `someFunc`, на экране появится предупреждающее сообщение, и сценарий не будет остановлен, а продолжит работать.

5.4.1. Генерирование исключений

Мы можем сгенерировать исключение искусственно, чтобы сообщить другому фрагменту сценария о возникшей нештатной ситуации.

> *Оператор генерирования исключения* `throw` принимает в качестве параметра *объект исключения*. Формат этого оператора:
>
> `throw <объект исключения>`

Для создания *объекта исключения* следует использовать оператор `new` (см. *разд. 4.6*). Объект создается на основе наиболее подходящего класса из упомянутых ранее.

Сгенерированное исключение может быть обработано посредством обычного обработчика.

Вот вариант функции `circleLength`, проверяющей, является ли переданный ей параметр числом, и, если это не так, генерирующей исключение класса `TypeError`:

```
function circleLength(d) {
    let l;
    if (typeof d != 'number') {
        const ex = new TypeError();
        ex.message = 'Допустимы только числа';
        throw ex;
    }
    l = d * Math.PI;
    return l;
}
```

Генерируемое этой функцией исключение мы можем обработать следующим образом:

```
window.document.write('<tr><td>yte</td>');
window.document.write('<td>');
try {
    // Пытаемся вызвать функцию circleLength, передав ей строку 'yte',
    // и вывести возвращенный ей результат на экран
    window.document.write(circleLength('yte'));
}
catch(ex) {
    // Если функция сгенерировала исключение, выводим сообщение об ошибке
    window.document.write(ex.message);
}
finally {
    // В любом случае выводим закрывающие теги
    window.document.write('</td></tr>');
}
```

Генерирование исключений часто применяется при проверке, правильные ли параметры были переданы функции, присутствует ли на странице нужный элемент и т. п.

5.5. Самостоятельные упражнения

- Найдите в папке 5\!sources сопровождающего книгу электронного архива (см. *приложение 3*) страницу 4.14.1.html и перепишите содержащийся в ней сценарий таким образом, чтобы он в случае нажатия кнопки **Отмена** выводил на экран соответствующее сообщение. Используйте для этого обработчик исключений.

- Перепишите сценарий из страницы 3.11.2.html, находящейся в той же папке архива, таким образом, чтобы функция `average`, вычисляющая среднее арифметическое, генерировала исключение класса `TypeError`, если ей не был передан ни один числовой параметр. В коде, вызывающем эту функцию, реализуйте обработку исключения с выводом сообщения, извлеченного из свойства `message` объекта исключения. Проверьте исправленную функцию на следующих наборах параметров:

```
average(10, 20, 30, 40, 50, 60, 70, 80, 90, 100);
average('10', '20', '30', '40', '50', '60', '70', '80', '90', '100')
average()
```

ЧАСТЬ II

Управление веб-страницей и веб-обозревателем

- События.
- DOM и BOM.
- Графика и мультимедиа.
- Веб-формы и элементы управления.
- Регулярные выражения.

Урок 6
События и их обработка

События. Обработчики событий
Понятие DOM
Получение сведений о событии
Фазы события
Обработчик по умолчанию и его отмена

6.1. Упражнение. Обрабатываем события

Напишем простой *слайдер* — элемент, выводящий на экран три разных изображения при щелчках на расположенных ниже трех кнопках.

1. В папке 6\!sources сопровождающего книгу электронного архива (см. *приложение 3*) найдем страницу 6.1.html, содержащую заготовку слайдера, и папку images с изображениями 1.jpg, 2.jpg и 3.jpg, которые будут выводиться с его помощью. Скопируем их в какую-либо папку на локальном диске.

   Страница 6.1.html с заготовкой для слайдера выглядит так, как показано на рис. 6.1. В ее нижней части находятся три кнопки с порядковыми номерами, выводящие на экран изображения. Изначально слайдер показывает изображение из

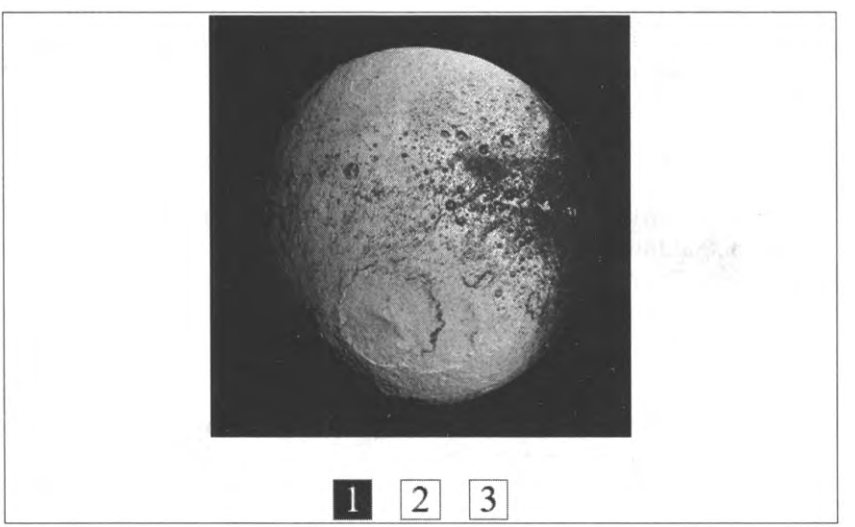


Рис. 6.1. Исходное состояние страницы 6.1.html

файла 1.jpg, а кнопка **1**, соответствующая выведенному изображению (активная), выделена инверсной цветовой гаммой.

2. Откроем файл 6.1.html в текстовом редакторе. Вот HTML-код, создающий сам слайдер:

```
<section>
    <img id="output" src="images/1.jpg">
</section>
<nav>
    <div id="1" class="active">1</div>
    <div id="2">2</div>
    <div id="3">3</div>
</nav>
```

Тег `<section>` создает панель, в которой будет показываться очередное изображение. Для его вывода применяется тег `<img>` с якорем `output`.

> *Якорь* — уникальная пометка, которая задается у тега в атрибуте `id`. По этой пометке в веб-сценарии мы сможем получить объект, представляющий этот тег, для манипулирования им.

Изначально выводится первое изображение — из файла 1.jpg.

Тег `<nav>` создает набор кнопок. Чаще всего кнопки слайдера формируются блоками (тегами `<div>`) или гиперссылками — их проще оформить с применением стилей. У всех кнопок также заданы якоря: `1`, `2` и `3`. Первая, активная, кнопка помечена стилевым классом `active`.

В коде страницы присутствует внутренняя таблица стилей, оформляющая все эти элементы нужным образом.

В сценарии, который мы напишем, придется взаимодействовать с объектами, представляющими «кнопки» `1`, `2` и `3` (возьмем название «кнопки» в кавычки, поскольку это не настоящие кнопки, создаваемые тегом `<input>`), а также элемент `output`. Все эти объекты входят в состав DOM.

> *DOM* (Document Object Model, объектная модель документа) — структура взаимосвязанных объектов, представляющих элементы страницы.

Чтобы сценарий, манипулирующий элементами страницы, смог получить доступ к представляющим их объектам, эти объекты к моменту выполнения сценария уже должны быть сформированы в памяти. Для этого сценарий помещают после HTML-кода, создающего объекты, лучше всего — после закрывающего тега `<html>`.

3. После закрывающего тега `<html>` запишем тег `<script>`, в который заключим код сценария (напомним, что добавления и правки в уже написанном коде здесь и далее выделены полужирным шрифтом):

```
<html>
    . . .
</html>
```

```
<script type="text/javascript">
</script>
```

Теперь нам нужно получить объекты DOM, представляющие тег <img>, в котором будут выводиться изображения, и блоки-«кнопки». Поскольку у тега <img> указан якорь output, мы можем найти соответствующий объект по этому якорю.

4. Вставим в тег <script> первое выражение сценария, которое присвоит константе output объект, представляющий элемент output:

```
const output = window.document.getElementById('output');
```

Для получения элемента страницы по его якорю применен метод getElementById класса HTMLDocument. Объект класса HTMLDocument, представляющий текущую страницу, хранится в свойстве document объекта window, который представляет текущее окно веб-обозревателя.

Примечание

Подробно объекты и классы DOM, их свойства и методы мы рассмотрим в *уроке 7*.

5. Чтобы получить коллекцию объектов, представляющих «кнопки», мы выполним поиск по их CSS-селектору nav div (блоки , вложенные в панель навигации <nav>).

С этой целью добавим выражение, присваивающее константе buttons коллекцию «кнопок»:

```
const buttons = window.document.querySelectorAll('nav div');
```

Метод querySelectorAll возвращает коллекцию элементов, соответствующих указанному CSS-селектору.

6. Нам также понадобится переменная для хранения объекта активной «кнопки». Объявим эту переменную, назвав ее current, и присвоим ей первую, активную, «кнопку»:

```
let current = buttons[0];
```

7. Нам нужно, чтобы при щелчках на «кнопках» 1, 2 и 3 в элемент output выводилось соответствующее изображение. Для этого надо написать обработчик события click и привязать его к «кнопкам».

> *Событие* — сигнал от веб-обозревателя о том, что в элементе страницы что-то произошло. Каждое событие, могущее возникнуть в элементе, имеет уникальное имя (например, событие щелчка мышью имеет имя click).

> *Обработчик события* — функция, выполняемая при возникновении определенного события в определенном элементе страницы. Может принимать один параметр — объект, хранящий сведения о возникшем событии.

Итак, объявим функцию showImage — обработчик события click:

```
function showImage() {
    current = this;
```

```
    output.src = 'images/' + this.id + '.jpg';
    buttons.forEach((el) => {
        if (el == this)
            el.className = 'active';
        else
            el.className = '';
    });
}
```

В теле обработчика события локальная переменная `this`, созданная автоматически, хранит объект элемента страницы, к которому привязан обработчик. В нашем случае таким элементом будет «кнопка», на которой щелкнули мышью.

Свойство `id` элемента страницы хранит его якорь. Мы формируем на его основе ссылку на графический файл и присваиваем ее свойству `src` элемента `output`, чтобы вывести этот файл на экран.

Далее помечаем «кнопку», на которой был выполнен щелчок, как активную, привязав к ней стилевой класс `active`, и убираем этот стилевой класс у всех остальных «кнопок». Коллекция, возвращенная методом `querySelectorAll`, поддерживает метод `forEach`, знакомый нам по массивам (см. *разд. 4.5*). Функция, указанная в вызове этого метода, проверяет, является ли очередная «кнопка» из коллекции той, на которой был выполнен щелчок, и, если это так, присваивает свойству `className`, хранящему привязанные к элементу стилевые классы, строку `'active'`, а в противном случае — пустую строку.

8. Осталось привязать только что написанный обработчик к событию `click`, возникающему во всех «кнопках» из коллекции `buttons`.

> Привязка обработчика события выполняется вызовом у нужного элемента страницы метода `addEventListener`. Его формат:
>
> `addEventListener(<`*имя события*`>, <`*обработчик*`>)`
>
> *Имя события* указывается в виде строки, а *обработчик* — в виде ссылки на нужную функцию.

Напишем выражение, которое выполнит привязку обработчика:

```
buttons.forEach(
    (el) => {
        el.addEventListener('click', showImage);
    }
);
```

Откроем страницу 6.1.html в веб-обозревателе и щелкнем на кнопке **3**. На странице появится третья картинка — взятая из файла 3.jpg (рис. 6.2).

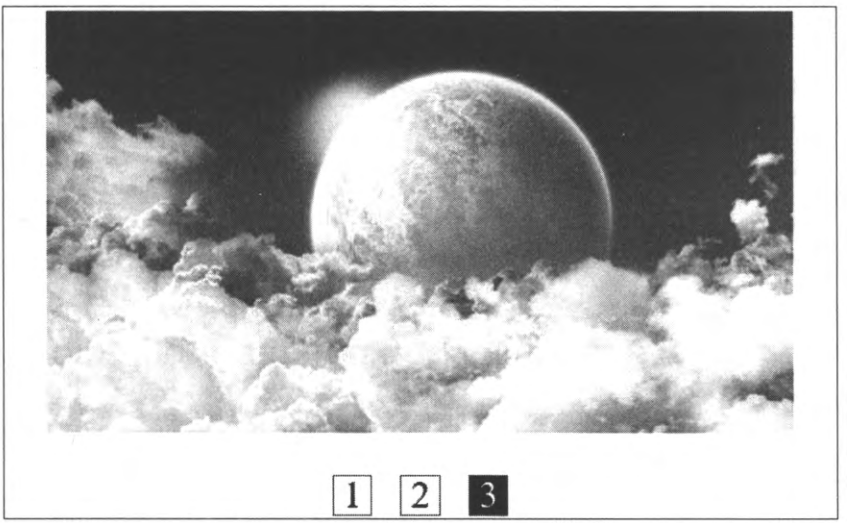


Рис. 6.2. Вид слайдера после щелчка на кнопке 3

6.1.1. Еще немного об обработчиках событий

- В качестве обработчика события, помимо именованной, могут быть использованы анонимная функция или функция-стрелка:

```
el.addEventListener('click', function() {
    . . .
});

el.addEventListener('click', () => {
    . . .
});
```

- В функциях-стрелках переменная `this`, хранящая элемент, к которому привязан обработчик, не создается, и обращение к ней вызовет ошибку:

```
el.addEventListener('click', () => {
    current = this;  // Ошибка!
    . . .
});
```

- К одному и тому же событию одного и того же элемента можно привязать сколько угодно обработчиков. Они будут выполнены в том порядке, в котором привязаны:

```
el.addEventListener('click', listener1);
el.addEventListener('click', listener2);
el.addEventListener('click', listener3);
```

6.1.2. Удаление привязки обработчика к событию

Удалить привязанный ранее обработчик события можно вызовом у нужного элемента страницы метода `removeEventListener`:

```
removeEventListener(<имя события>, <обработчик>)
```

Пример:

```
el.removeEventListener('click', listener2);
```

6.1.3. Альтернативный способ привязки обработчиков к событиям

С помощью этого способа обработчик можно привязать:

- в HTML-коде — использованием атрибута тега с именем вида `on<имя события>`, в качестве значения которого указывается непосредственно *код обработчика* (например, вызов функции):

  ```
  <div id="1" onclick="showImage();">1</div>
  ```

  Сценарий, объявляющий функцию-обработчик, может располагаться после тега, к которому привязан этот обработчик (поскольку он в любом случае будет выполняться после загрузки всего кода страницы):

  ```
  <div id="1" onclick="showImage();">1</div>
  . . .
  <script type="text/javascript">
      function showImage() {
          . . .
      }
  </script>
  ```

  В этом случае функция-обработчик, привязанная в HTML-коде, не получит в качестве параметра объект события (с которым мы познакомимся в *разд. 6.3*) и, соответственно, не сможет узнать сведения о событии;

- в коде сценария — с применением свойства, которое имеет такое же имя, как и упомянутый ранее атрибут тега, и которому присваивается *сама функция-обработчик*:

  ```
  const div1 = window.document.getElementById('1');
  div1.onclick = showImage;
  ```

Альтернативный способ рекомендуется применять только для привязки событий к секции тела страницы (тегу `<body>`), т. к. привязка методом `addEventListener` в ее случае не срабатывает.

- Вот так это можно выполнить в HTML-коде:

  ```
  <body onkeydown="processKeyDown();">
  ```

♦ А так — в коде сценарии, чисто программно:

```
const body = window.document.body;
body.onkeydown = processKeyDown;
```

В остальных случаях альтернативный способ применять не рекомендуется, поскольку он считается устаревшим.

6.2. События, поддерживаемые элементами страницы

Элементы страницы поддерживают весьма много событий. В табл. 6.1 приведены только основные.

Таблица 6.1. *Основные события, поддерживаемые элементами страницы*

Имя события	Когда возникает?
События мыши	
click	Щелчок мышью
dblclick	Двойной щелчок мышью
contextmenu	Щелчок правой кнопкой мыши перед выводом контекстного меню
mousedown	Нажатие кнопки мыши
mouseup	Отпускание кнопки мыши
mouseenter	Наведение курсора мыши на элемент страницы*
mouseover	
mousemove	Перемещение курсора мыши на элементе страницы
mouseleave	Увод курсора мыши с элемента страницы*
mouseout	
wheel	Прокрутка колесика мыши
События клавиатуры Возникают только в элементах, способных принимать клавиатурный ввод (в гиперссылках и элементах управления), а также в секции тела страницы (теге <body>)	
keydown	Нажатие любой клавиши
keypress	Нажатие алфавитно-цифровой клавиши
keyup	Отпускание нажатой ранее клавиши
Событие буфера обмена	
copy	Копирование выделенного текста
События CSS-анимации	
transitionend	Конец воспроизведения анимации с двумя состояниями (создаваемой атрибутами стиля семейства transition)

Таблица 6.1 (окончание)

Имя события	Когда возникает?
animationstart	Начало воспроизведения анимации с несколькими состояниями (создаваемой атрибутами стиля семейства animation)
animationiteration	Завершено очередное повторение анимации с несколькими состояниями (если она воспроизводится только один раз, не возникает)
animationend	Конец воспроизведения анимации с несколькими состояниями
Прочие события	
scroll	Прокрутка содержимого элемента страницы (если это элемент с прокруткой)
События тела страницы Возникают только в секции тела страницы (теге <body>). Обработчики к этим событиям следует привязывать только с помощью альтернативного способа (см. *разд. 6.1.3*)	
load	После загрузки страницы
pageshow	После загрузки страницы и события load
resize	При изменении размеров страницы (что происходит при изменении размеров окна веб-обозревателя, в котором она открыта)
beforeunload	Перед уходом с текущей страницы (что может быть вызвано переходом на другую страницу или закрытием текущей вкладки)
pagehide	Перед уходом с текущей страницы, после события beforeunload
beforeprint	Перед открытием стандартного диалогового окна печати страницы
afterprint	После закрытия стандартного диалогового окна печати страницы нажатием кнопки **OK** или **Отмена**
hashchange	При переходе на другой якорь

* Эти два события возникают при одинаковых условиях, но отличаются по количеству фаз их «прохождения» (подробно о фазах событий рассказано в *разд. 6.4*).

Чаще всего обрабатываются события мыши и клавиатуры. Остальные события, в особенности события тела страницы, обрабатываются много реже и в специальных случаях.

6.2.1. Особенности события *beforeunload*

Если обработчик события beforeunload возвращает строку, то при уходе с текущей страницы появится стандартное окно-предупреждение с кнопками **Закрыть** и **Отмена**. Уход со страницы будет выполнен только при нажатии кнопки **Закрыть**:

```
<body onbeforeunload="return 'Покинуть эту страницу?'">
    . . .
    <p><a href="https://www.w3.org/">На сайт W3C</a></p>
    . . .
</body>
```

6.3. Упражнение. Получаем сведения о событии

Сделаем так, чтобы при наведении курсора мыши на одну из «кнопок» слайдера, рассмотренного в *разд. 6.1*, на экране рядом с курсором появлялась миниатюра соответствующего изображения.

Реализовать это можно в обработчиках событий `mouseover` и `mouseout`: первый обработчик выведет миниатюру на экран, второй скроет ее. А получить координаты курсора мыши можно из объекта события, который передается любой функции-обработчику с единственным параметром (об этом говорилось в *разд. 6.1*).

1. Итак, продолжим работу над текстом страницы 6.1.html и добавим после панели навигации (`<nav>`) HTML-код, создающий блок, в котором будет выводиться миниатюра :

```
<nav>
    . . .
</nav>
<div id="preview"></div>
```

   Чтобы позже получить объект, представляющий этот блок, пометим его якорем `preview`.

2. Добавим во внутреннюю таблицу стилей слайдера стили, задающие оформление для этого блока:

```
#preview {
    position: absolute;
    width: 120px;
    height: 80px;
    background: left/contain no-repeat;
    display: none;
}
#preview.active {
    display: block;
}
```

   Миниатюру в этом блоке мы будем выводить в виде графического фона — так проще.

3. В начало кода сценария добавим выражение, которое получит объект, представляющий блок `preview`:

```
const preview = window.document.getElementById('preview');
```

4. Там же объявим функцию `getImageURL`, которая примет в качестве параметра имя графического файла и вернет полную ссылку на него:

```
function getImageURL(fileName) {
    return 'images/' + fileName + '.jpg'
}
```

5. Исправим код обработчика события `showImage` так, чтобы для формирования ссылки на файл он использовал функцию `getImageURL`:

```
function showImage() {
    current = this;
    output.src = getImageURL(this.id);
    . . .
}
```

6. Объявим функцию `showPreview` — обработчик события `mouseover` «кнопок», которая выведет на экран блок `preview` с миниатюрой графического изображения, соответствующего «кнопке», на которую был наведен курсор мыши:

```
function showPreview(evt) {
    preview.style.left = evt.pageX + 'px';
    preview.style.top = evt.pageY + 'px';
    preview.style.backgroundImage = 'url(' +
        getImageURL(this.id) + ')';
    preview.className = 'active';
}
```

Параметр `evt` будет хранить объект класса `MouseEvent`, представляющий событие (более обстоятельный разговор о нем пойдет чуть позже). Свойства `pageX` и `pageY` этого класса хранят горизонтальную и вертикальную координаты курсора мыши относительно левого верхнего угла страницы в пикселах. Добавив к этим величинам обозначение единицы измерения — `px` (пикселы в языке CSS), — мы получим готовые координаты блока `preview`. Далее мы выводим соответствующее «кнопке» изображение в качестве графического фона этого блока, после чего привязываем к нему стилевой класс `active`, чтобы вывести его на экран.

7. Объявим функцию `hidePreview` — обработчик события `mouseout` «кнопок», которая скроет блок `preview`:

```
function hidePreview() {
    preview.className = '';
}
```

8. Наконец, дополним код, привязывающий обработчики к событиям, двумя выражениями, которые выполнят привязку написанных ранее обработчиков событий `mouseover` и `mouseout`:

```
buttons.forEach(
    (el) => {
        el.addEventListener('click', showImage);
        el.addEventListener('mouseover', showPreview);
        el.addEventListener('mouseout', hidePreview);
    }
);
```

Откроем страницу 6.1.html в веб-обозревателе, наведем курсор мыши на кнопку **3** и полюбуемся миниатюрой, появившейся под курсором мыши (рис. 6.3).

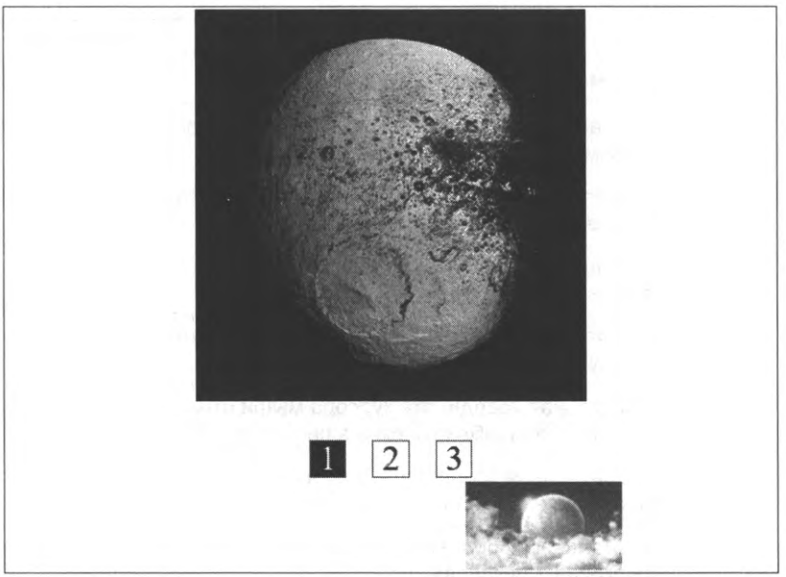


Рис. 6.3. При наведении курсора мыши на кнопку 3 появилась миниатюра рисунка

6.3.1. Классы событий и их свойства

Различные события представляются объектами разных классов. В табл. 6.2 приведены основные классы событий и их свойства.

Примечание

Все свойства классов событий доступны только для чтения значений. Попытка присвоить такому свойству новое значение вызовет ошибку.

Таблица 6.2. *Основные классы событий и их свойства*

Свойство	Хранимое значение
Класс `Event` Представляет все события, кроме приведенных далее. Также является базовым для всех остальных классов событий	
`currentTarget`*	Элемент страницы, к которому привязан текущий обработчик (то же, что и переменная `this`)
`target`*	Элемент страницы, в котором изначально возникло событие (*источник события*)
`type`	Имя возникшего события в виде строки (например, `'click'` или `'mouseover'`). Полезно, если один и тот же обработчик применяется для обработки разных событий.
Класс `MouseEvent` Производный от класса `Event`. Представляет все события мыши, кроме `wheel`	
`pageX`	Горизонтальная координата курсора мыши относительно левого верхнего угла самой страницы в пикселах

***Таблица 6.2** (продолжение)*

Свойство	Хранимое значение
pageY	Вертикальная координата курсора мыши относительно левого верхнего угла самой страницы в пикселах
offsetX	Горизонтальная координата курсора мыши относительно левого верхнего угла элемента-источника в пикселах
offsetY	Вертикальная координата курсора мыши относительно левого верхнего угла элемента-источника в пикселах
clientX	Горизонтальная координата курсора мыши относительно левого верхнего угла внутренней области окна в пикселах
clientY	Вертикальная координата курсора мыши относительно левого верхнего угла внутренней области окна в пикселах
screenX	Горизонтальная координата курсора мыши относительно левого верхнего угла экрана в пикселах
screenY	Вертикальная координата курсора мыши относительно левого верхнего угла экрана в пикселах
button	Числовое обозначение нажатой кнопки мыши: 0 — левая, 1 — средняя или колесико, 2 — правая
which	Числовое обозначение нажатой кнопки мыши: 0 — ни одна кнопка не была нажата, 1 — левая, 2 — средняя или колесико, 3 — правая
ctrlKey	true, если в текущий момент нажата клавиша <Ctrl>, и false в противном случае
shiftKey	true, если в текущий момент нажата клавиша <Shift>, и false в противном случае
altKey	true, если в текущий момент нажата клавиша <Alt>, и false в противном случае
relatedRatget	В случае события mouseover или mouseenter — элемент страницы, с которого был уведен курсор мыши. В случае события mouseout или mouseleave — элемент, на который был наведен курсор
Класс WheelEvent Производный от класса Event. Представляет событие wheel	
deltaY	Расстояние прокрутки по вертикали: положительное, если выполнялась прокрутка вниз, и отрицательное в случае прокрутки вверх
deltaX	Расстояние прокрутки по горизонтали: положительное, если выполнялась прокрутка вправо, и отрицательное в случае прокрутки влево. Если мышь не поддерживает прокрутку по горизонтали, всегда хранит 0
deltaMode	Обозначение единицы измерения для расстояния прокрутки: 0 — пикселы, 1 — строки, 2 — страницы
Класс KeyboardEvent Производный от класса Event. Представляет события клавиатуры	
charCode	В случае события keypress — код введенного символа**. В случае событий keydown и keyup — всегда 0

Таблица 6.2 (окончание)

Свойство	Хранимое значение
which	В случае события keypress — то же, что и charCode. В случае событий keydown и keyup — Unicode-код нажатой клавиши***
key	Наименование нажатой клавиши в виде строки (например: 'a', 'B', 'Enter')
ctrlKey	true, если в текущий момент нажата клавиша <Ctrl>, и false в противном случае
shiftKey	true, если в текущий момент нажата клавиша <Shift>, и false в противном случае
altKey	true, если в текущий момент нажата клавиша <Alt>, и false в противном случае
Класс ClipboardEvent Производный от класса Event. Представляет события буфера обмена	
Класс TransitionEvent Производный от класса Event. Представляет событие transitionend	
propertyName	Имя анимируемого атрибута стиля в виде строки
elapsedTime	Продолжительность анимации в виде числа в секундах
Класс AnimationEvent Производный от класса Event. Представляет события animationstart, animationiteration и animationend	
animationName	Имя набора состояний анимации в виде строки
elapsedTime	Продолжительность анимации в виде числа в секундах
Класс PageTransitionEvent Производный от класса Event. Представляет события pageshow и pagehide	
persisted	true — если страница была сохранена в кэше, и false в противном случае
Класс HashChangeEvent Производный от класса Event. Представляет событие hashchange	
oldURL	Интернет-адрес, с которого был выполнен переход
newURL	Интернет-адрес, на который был выполнен переход

* Свойства currentTarget и target хранят одинаковые значения только на фазе источника (подробно о фазах событий рассказано в *разд. 6.4*).

** Таблицу Unicode-кодов символов можно найти на странице: **https://www.w3schools.com/charsets/ref_html_utf8.asp**.

*** Коды клавиш клавиатуры приведены на странице: **http://umi-cms.spb.su/ref/javascript/251/**.

6.4. Упражнение. Изучаем фазы событий

Сценарий в коде страницы 6.1.html неоптимален: мы привязали обработчики событий к каждой «кнопке» слайдера, что привело к избыточной трате системных ресурсов. Оптимальнее привязать их к панели навигации, в которой находятся «кнопки», и обрабатывать возникающие в «кнопках» события на фазе всплытия.

Каждое событие при возникновении последовательно проходит три *фазы*:

- *погружения* — событие сначала возникает в самом внешнем элементе (теге `<html>`), далее — в секции тела страницы (`<body>`), потом — наиболее «внешнем» родителе элемента-источника и т. д., пока не возникнет в непосредственном родителе источника;
- *источника* — событие возникает непосредственно в источнике;
- *всплытия* — событие возникает в непосредственном родителе источника, далее — родителе родителя и т. д., пока не возникнет в теге `<html>`.

Этот процесс носит название «*прохождения*» событий (event propagation).

Если щелкнуть мышью на одной из «кнопок» слайдера на странице 6.1.html, событие `click` пройдет по следующему пути:

№№	Элемент страницы	Фаза
1	`<html>`	Погружения
2	`<body>`	
3	`<nav>`	
4	`<div>`	Источника
5	`<nav>`	Всплытия
6	`<body>`	
7	`<html>`	

Событие можно обрабатывать в любом из родителей элемента-источника — на фазе всплытия или погружения. Для привязки обработчика в этом случае используется расширенный формат метода `addEventListener`:

```
addEventListener(<имя события>, <обработчик>
[, <фаза, на которой выполняется обработка>])
```

Если третьим параметром передать значение `false` или вообще не указывать его, обработка будет выполнена на *фазе всплытия*. Если же передать значение `true`, обработчик сработает на *фазе погружения*.

Фазы погружения и источника проходят все события.

Фазу всплытия проходят лишь *всплывающие* события:

- мыши, за исключением `mouseenter` и `mouseleave`;
- клавиатуры;

- буфера обмена;
- анимации.

Остальные события — *невсплывающие* — не проходят фазу всплытия.

Метод `removeEventListener` также поддерживает расширенный формат вызова — с указанием фазы:

```
removeEventListener(<имя события>, <обработчик>
[, <фаза, на которой выполняется обработка>])
```

1. Продолжим доработку кода страницы 6.1.html и зададим у панели навигации (`<nav>`) якорь `buttonset`:

```
<nav id="buttonset">
    . . .
</nav>
```

2. Вставим в начало сценария выражение, присваивающее панель навигации переменной `buttonset`:

```
const buttonset = window.document.getElementById('buttonset');
```

3. Исправим функцию `showImage` следующим образом:

```
function showImage(evt) {
    if (evt.target != this) {
        current = evt.target;
        output.src = getImageURL(evt.target.id);
        buttons.forEach((el) => {
            . . .
        });
    }
}
```

Нам нужно выводить картинку только в том случае, когда пользователь щелкнул на «кнопке». Проверить факт щелчка именно на «кнопке» просто: нужно лишь сравнить значения свойства `target` объекта-события (оно, как мы помним, содержит источник события) и переменной `this` (элемент, к которому привязан обработчик) — это должны быть разные объекты (если же они равны, то пользователь щелкнул на самой панели навигации, и мы ничего не делаем).

И не забываем, что раз обработчик привязан не к «кнопке», для получения доступа к «кнопке» нужно использовать свойство `target` объекта-события, а не переменную `this`.

4. Исправим код функций `showPreview` и `hidePreview`:

```
function showPreview(evt) {
    if (evt.target != this) {
        . . .
        preview.style.backgroundImage = 'url(' +
            getImageURL(evt.target.id) + ')';
```

```
            preview.className = 'active';
        }
    }
    function hidePreview(evt) {
        if (evt.target != this)
            preview.className = '';
    }
```

Проверим переделанную страницу в действии и убедимся, что все работает.

> Обработка событий на фазе погружения применяется относительно редко. Обычно в таких обработчиках реализуются всевозможные предварительные проверки и, если таковые не выполняются, дальнейшее «прохождение» события прерывается.

6.4.1. Свойства и методы класса *Event*, имеющие отношение к «прохождению» событий

Класс `Event` поддерживает два свойства, позволяющие выяснить детали «прохождения» события:

- `eventPhase` — возвращает числовое обозначение текущей фазы: `1` — погружения, `2` — источника, `3` — всплытия;
- `bubbles` — возвращает `true`, если событие всплывающее, и `false` — если оно невсплывающее.

Не принимающий параметров метод `stopPropagation` прерывает дальнейшее «прохождение» события:

```
function listener(evt) {
    . . .
    evt.stopPropagation();
}
```

6.5. Упражнение. Отменяем обработку событий по умолчанию

Блок (тег `<div>`) — не лучший выбор для создания «кнопок» слайдера. Удобнее делать их на основе гиперссылок: тег `<a>` поддерживает атрибут `href`, в котором можно записать полную ссылку на нужный файл. Давайте заменим в нашем слайдере блоки на гиперссылки.

1. Снова обратимся к коду страницы 6.1.html и исправим теги `<div>` на `<a>`, а также запишем ссылки на файлы в атрибутах `href` этих тегов:

```
<a href="images/1.jpg" class="active">1</a>
<a href="images/2.jpg">2</a>
<a href="images/3.jpg">3</a>
```

2. Исправим селекторы в стилях, оформляющих «кнопки», и доделаем первый стиль:

```
nav a {
    . . .
    display: block;
    text-decoration: none;
}
nav a.active {
    . . .
}
```

3. Внесем правки в код сценария:

```
. . .
const buttons = window.document.querySelectorAll('nav a');
. . .
function showImage(evt) {
    if (evt.target != this) {
        . . .
        output.src = evt.target.href;
        . . .
    }
}
function showPreview(evt) {
    if (evt.target != this) {
        . . .
        preview.style.backgroundImage = 'url(' +
            evt.target.href + ')';
        . . .
    }
}
```

Свойство `href` соответствует одноименному атрибуту тега `<a>`.

4. Функция `getImageURL` нам теперь не нужна, так что удалим ее:

```
function getImageURL(fileName) {
    . . .
}
```

Откроем исправленную страницу 6.1.html в веб-обозревателе и щелкнем на кнопке **3**. Увы — изображение 3.jpg будет открыто, но не в панели слайдера, а непосредственно в веб-обозревателе, заменив собой всю страницу (рис. 6.4).

Проблема в том, что при щелчке на гиперссылке выполнился обработчик по умолчанию, исполнивший переход по этой гиперссылке.

> *Обработчик события по умолчанию* — реализован в самом веб-обозревателе и срабатывает после всех обработчиков, написанных программистом.

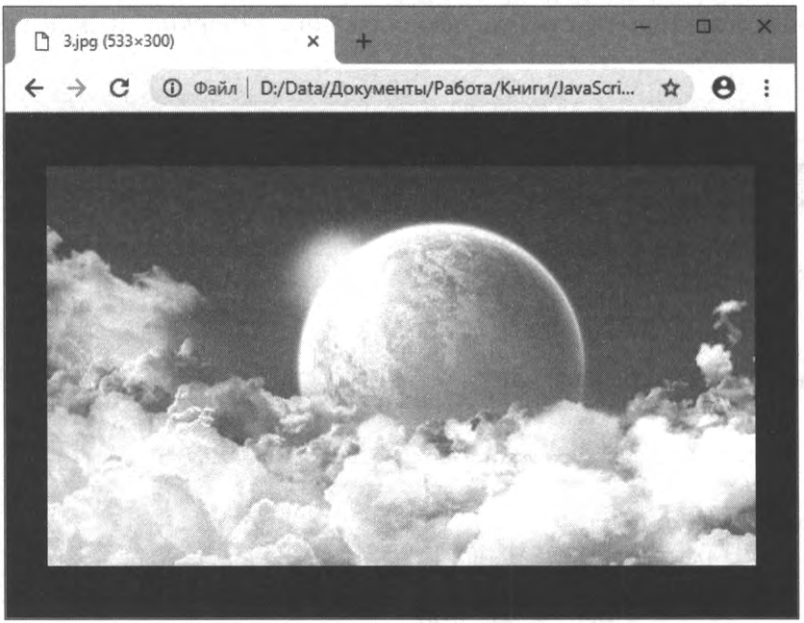


Рис. 6.4. Изображение 3.jpg открыто не в панели слайдера, а непосредственно в окне веб-обозревателя

Обработчик по умолчанию у разных событий выполняет разные действия:

- `click` — зависит от элемента страницы (у гиперссылок — переход на целевую страницу, у кнопок — нажатие);
- `contextmenu` — вывод контекстного меню;
- события клавиатуры — обработка нажатой клавиши;
- `copy` — копирование текста в буфер обмена.

У остальных событий обработчики по умолчанию отсутствуют.

Отменить обработчик по умолчанию можно, вызвав у объекта события не принимающий параметров метод `preventDefault`, который поддерживается классом `Event`. Вызов метода выполняется в теле обработчика события.

5. Отменим для нашей страницы 6.1.html обработчик события по умолчанию, вставив в конец тела функции `showImage` вызов метода `preventDefault`:

```
function showImage(evt) {
    if (evt.target != this) {
        . . .
        evt.preventDefault();
    }
}
```

Вот теперь наш слайдер будет работать, как положено.

6.5.1. Свойства класса *Event*, касающиеся обработки события по умолчанию

Класс `Event` поддерживает два свойства, позволяющие получить сведения об обработчике события по умолчанию:

- `cancellable` — возвращает `true`, если обработку события по умолчанию можно отменить, и `false` в противном случае;
- `defaultPrevented` — возвращает `true`, если обработчик по умолчанию этого события был отменен ранее, и `false` в противном случае.

6.6. Самостоятельные упражнения

- В сценарии на странице 6.1.html сделайте так, чтобы при щелчке правой кнопкой мыши на элементе `output` не выводилось контекстное меню (чтобы предотвратить сохранение изображений на локальном диске).

  Подсказка: привяжите к событию `contextmenu` обработчик, в котором отменяйте обработку события по умолчанию.

- Там же сделайте так, чтобы при наведении курсора мыши на активную «кнопку» миниатюра не выводилась (все равно она не нужна).

  Подсказка: обрабатывайте событие `mouseover` в панели навигации на фазе погружения и в теле обработчика прерывайте дальнейшее «прохождение» события.

- Там же реализуйте переход от одного изображения к другому нажатием клавиш <стрелка влево> и <стрелка вправо>.

  Подсказка: обрабатывайте событие `keydown` в секции тела страницы (в теге `<body>`).

- Возьмите из папки 6\!sources сопровождающего книгу электронного архива (см. *приложение 3*) страницу 6.6.html с полем ввода. Напишите сценарий, который позволит вводить в это поле только цифры 0–9.

Урок 7
Управление элементами веб-страниц

Доступ к веб-странице и ее элементам
Управление элементами веб-страниц
Управление стилями
Создание новых элементов
Получение сведений о веб-странице

7.1. Получение доступа к элементам страницы

Прежде чем выполнять какие-либо действия над элементом страницы, следует получить представляющий его объект. Этот объект создается веб-обозревателем в процессе загрузки страницы и входит в набор объектов DOM (объектной модели документа).

Получить доступ к объектам элементов страницы можно только через объект, представляющий саму страницу.

При исполнении сценария веб-обозреватель создает переменную `window`, в которой сохраняет объект класса `Window`, представляющий текущее окно. Свойство `document` объекта окна хранит объект класса `HTMLDocument`, представляющий текущую веб-страницу, — он-то нам и нужен.

7.1.1. Доступ к элементам определенного типа

Класс `HTMLDocument` поддерживает ряд свойств, посредством которых можно получить доступ к некоторым (не всем!) элементам страницы:

- `body` — объект класса `HTMLBodyElement`, представляющий секцию тела страницы (тег `<body>`):

```
const body = window.document.body;
body.onkeydown = processKeyDown;
```

- `head` — объект класса `HTMLHeadElement`, представляющий секцию заголовка страницы (тег `<head>`);
- `images` — коллекция всех графических изображений (тегов `<img>`), что есть на странице:

```
let imageCount = window.document.images.length;
```

- `links` — коллекция всех гиперссылок (тегов `<a>`) и областей в картах-изображениях (тегов `<area>`);
- `forms` — коллекция всех веб-форм (тегов `<form>`);
- `scripts` — коллекция всех веб-сценариев, привязанных к странице;
- `activeElement` — элемент страницы, имеющий фокус клавиатурного ввода (клавиатурный ввод могут принимать лишь гиперссылки, элементы управления и секция тела страницы — тег `<body>`).

7.1.2. Доступ к любому элементу страницы

Следующие методы класса `HTMLDocument` позволят получить доступ к любому элементу страницы:

- `getElementById(<якорь>)` — возвращает элемент с указанным *якорем* (задается в атрибуте тега `id`) или `null`, если такого элемента нет:

```
<img id="output" src="images/1.jpg">
. . .
const output = window.document.getElementById('output');
```

- `getElementsByName(<наименование>)` — возвращает коллекцию элементов с указанным *наименованием* (задается в атрибуте тега `name`). Если таковых элементов нет, возвращает пустую коллекцию:

```
<input type="text" name="name1">
. . .
const name1 = window.document.getElementsByName('name1')[0];
```

  Возвращаемый методом объект коллекции относится к классу `NodeList`. Этот класс поддерживает метод `forEach` (см. *разд. 4.5*);

- `getElementsByTagName(<имя тега>)` — возвращает коллекцию элементов, созданных тегом с указанным *именем* (ставится без символов < и >). Если таковых элементов нет, возвращает пустую коллекцию:

```
<ul>
    <li>HTML;</li>
    <li>CSS;</li>
    <li>JavaScript.</li>
</ul>
. . .
// Получаем второй по счету пункт списка
const li2 = window.document.getElementsByTagName('li')[1];
```

  Коллекция, возвращаемая этим методом, относится к классу `HTMLCollection`, который не поддерживает метод `forEach`;

- `getElementsByClassName(<имя стилевого класса>)` — возвращает коллекцию элементов (в виде объекта класса `HTMLCollection`), к которым был привязан стиле-

вой класс с указанным *именем* (ставится без начальной точки). Если таковых элементов нет, возвращает пустую коллекцию:

```
<div class="button">1</div>
<div class="button">2</div>
<div class="button">3</div>
. . .
const buttons = window.document.getElementsByClassName('button');
```

Можно указать несколько *имен* стилевых классов, разделив их пробелами, — тогда в полученной коллекции окажутся элементы, к которым были привязаны все эти стилевые классы:

```
<div class="button active">1</div>
<div class="button">2</div>
<div class="button">3</div>
. . .
let activeButton = window.document.↵
getElementsByClassName('button active')[0];
```

- `querySelector(<CSS-селектор>)` — возвращает первый имеющийся на странице элемент, соответствующий заданному *CSS-селектору*, или `null`, если подходящего элемента нет:

```
<nav>
    <div id="1" class="active">1</div>
    <div id="2">2</div>
    <div id="3">3</div>
</nav>
. . .
// Получаем последнюю "кнопку" слайдера
const lastButton = window.document.↵
querySelector('nav div:last-child');
```

- `querySelectorAll(<CSS-селектор>)` — возвращает коллекцию элементов (объект класса `NodeList`), соответствующих указанному *CSS-селектору*. Если таковых элементов нет, возвращает пустую коллекцию:

```
const buttons = window.document.querySelectorAll('nav div');
```

Можно указать несколько CSS-селекторов, разделив их запятыми, — тогда в коллекции окажутся элементы, соответствующие всем заданным селекторам:

```
// Получаем коллекцию из панели навигации и вложенных в нее "кнопок"
const navAndButtons =
    window.document.querySelectorAll('nav, nav div');
```

Объект, представляющий элемент страницы, относится к классу, который представляет HTML-тег, создающий этот элемент. Так, абзацы (теги `<p>`) представляются объектами класса `HTMLParagraphElement`, заголовки (теги `<h1>` . . . `<h6>`) — объектами класса `HTMLHeadingElement`, а блоки (теги `<div>`) — объектами класса `HTMLDivElement`. Все классы-теги являются производными от класса `HTMLElement`.

Класс `HTMLElement` поддерживает все упомянутые ранее методы, за исключением `getElementById` и `getElementsByName`. Так что мы можем искать нужные элементы внутри какого-либо элемента:

```
<nav>
    <div id="1" class="active">1</div>
    <div id="2">2</div>
    <div id="3">3</div>
</nav>
. . .
// Получаем панель навигации
const nav = window.document.getElementsByTagName('nav')[0];
// И ищем в ней активную "кнопку"
activeButton = nav.querySelector('div.active');
```

7.1.3. Доступ к родителю, соседям и потомкам

Класс `HTMLElement` поддерживает свойства, позволяющие добраться до родителя, соседей и потомков текущего элемента:

- `parentElement` — родитель текущего элемента или `null`, если текущий элемент не имеет родителя:

```
// Получаем родитель панели навигации
const body = nav.parentElement;  // Тег <body>
```

- `children` — коллекция потомков текущего элемента в виде объекта класса `HTMLCollection`:

```
// Получаем коллекцию "кнопок" слайдера
const buttons = nav.children;
```

- `childElementCount` — количество потомков текущего элемента;
- `firstElementChild` — первый потомок текущего элемента или `null`, если текущий элемент не имеет потомков:

```
// Получаем первую "кнопку" слайдера
const firstButton = nav.firstElementChild;
```

- `lastElementChild` — последний потомок текущего элемента или `null`, если текущий элемент не имеет потомков;
- `nextElementSibling` — следующий сосед текущего элемента или `null`, если такового нет;
- `previousElementSibling` — предыдущий сосед текущего элемента или `null`, если такового нет.

Также поддерживается метод `contains(<элемент>)`, который возвращает `true`, если указанный *элемент* является потомком текущего элемента, и `false` в противном случае:

```
// Проверяем, является ли активная "кнопка" потомком панели навигации
let result = nav.contains(activeButton);  // true
```

7.1.4. Контекст исполнения веб-сценариев

Код сценария выполняется веб-обозревателем «внутри» (как говорят программисты, в *контексте исполнения*) объекта класса `Window`, представляющего текущее окно и хранящегося в переменной `window`. При этом:

- при обращении к переменной по ее имени сначала выполняется поиск свойства объекта `window` с указанным именем. Если свойство найдено, выполняется обращение к нему, в противном случае ищется одноименная переменная;
- при вызове функции сначала ищется метод окна с таким же именем, и лишь в случае его отсутствия — одноименная функция.

Следовательно, на практике мы можем опустить обращение к переменной `window`. Так, если ранее мы писали:

```
const body = window.document.body;
const output = window.document.getElementById('output');
const buttons = window.document.querySelectorAll('nav a');
```

то сейчас можем записать так (несколько сократив код):

```
const body = document.body;
const output = document.getElementById('output');
const buttons = document.querySelectorAll('nav a');
```

В дальнейшем мы будем использовать именно такую запись.

7.2. Управление элементами веб-страницы

Для работы с элементами страниц класс `HTMLElement` предоставляет ряд свойств, которые можно разделить на три группы: соответствующие атрибутам тега, хранящие местоположение и размеры элемента и прочие.

Для представления атрибутов тега служат следующие свойства (табл. 7.1).

***Таблица 7.1.** Свойства класса `HTMLElement`, служащие для представления атрибутов тега*

Атрибут тега	Соответствующее свойство
`class`	`className`
`rowspan`	`rowSpan`
`colspan`	`colSpan`
Остальные	Имя свойства совпадает с именем атрибута тега

Все эти свойства непосредственно хранят значение соответствующего атрибута тега. Примеры:

```
// Привязываем к панели навигации стилевой класс navigation-panel,
// присвоив строку с его именем свойству className,
// которое соответствует атрибуту тега class
nav.className = 'navigation-panel';
// Создаем там же встроенный стиль, используя свойство style,
// что соответствует одноименному атрибуту стиля.
man.style = 'background-color: yellow;';

// Извлекаем из свойства id якорь и формируем на его основе ссылку
// на графический файл
let url = 'images/' + activeButton.id + '.jpg';
// Заносим полученную ссылку в свойство src элемента output,
// чтобы вывести изображение из файла на экран
output.src = url;
```

Исключение составляют свойства, представляющие атрибуты тегов без значений. Такое свойство хранит значение `true`, если соответствующий атрибут присутствует в теге, и `false` — в противном случае:

```
<input type="text" id="txtDesc">
. . .
const txtDesc = document.getElementById('txtDesc');
// Делаем поле ввода недоступным, присвоив значение true
// свойству disabled, которое соответствует одноименному
// атрибуту тега без значения
txtDesc.disabled = true;
```

Приведенные далее свойства местоположения и размеров хранят величины в пикселах, и все — за двумя исключениями — доступны только для чтения:

- `clientWidth` — видимая ширина текущего элемента, включая величины внутренних просветов слева и справа, но без учета рамки, ширины вертикальной полосы прокрутки и внешних просветов;
- `clientHeight` — видимая высота текущего элемента, включая величины внутренних просветов сверху и снизу, но без учета рамки, высоты горизонтальной полосы прокрутки и внешних просветов;
- `offsetLeft` — горизонтальная координата левого верхнего угла текущего элемента относительно левого верхнего угла элемента из свойства `offsetParent`;
- `offsetTop` — вертикальная координата левого верхнего угла текущего элемента относительно левого верхнего угла элемента из свойства `offsetParent`;
- `offsetParent` — элемент, относительно которого вычисляются значения свойств `offsetLeft` и `offsetTop`, — ближайший позиционируемый родитель или, если такового нет, секция тела страницы;
- `offsetWidth` — то же самое, что и `clientWidth`, но с учетом толщины левой и правой сторон рамки и ширины вертикальной полосы прокрутки;

- `offsetHeight` — то же самое, что и `clientHeight`, но с учетом толщины верхней и нижней сторон рамки и высоты горизонтальной полосы прокрутки;
- `scrollWidth` — ширина содержимого текущего элемента, включая величины внутренних просветов слева и справа, но без учета рамки, ширины вертикальной полосы прокрутки и внешних просветов;
- `scrollHeight` — высота содержимого текущего элемента, включая величины внутренних просветов сверху и снизу, но без учета рамки, ширины вертикальной полосы прокрутки и внешних просветов;
- `scrollLeft` — величина, на которую содержимое элемента прокручено по горизонтали. Доступно для чтения и записи;
- `scrollTop` — величина, на которую содержимое элемента прокручено по вертикали. Доступно для чтения и записи.

Понять назначение свойств «семейств» `offset` и `scroll` поможет рис. 7.1, изображающий элемент с прокруткой.

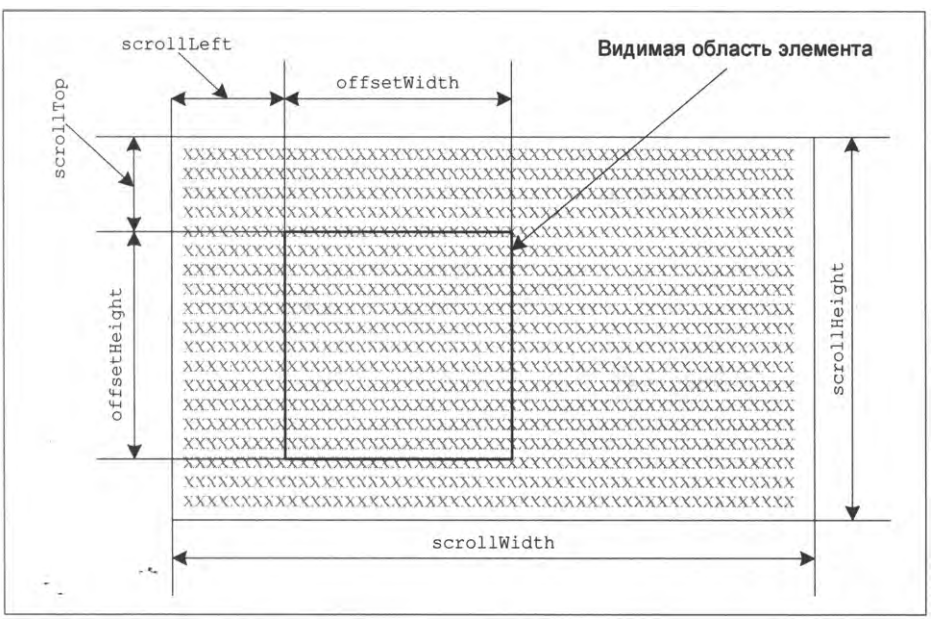


Рис. 7.1. Назначение свойств «семейств» `offset` **и** `scroll`

Прочие свойства немногочисленны:

- `textContent` — текстовое содержимое текущего элемента:

```
// Задаем для активной "кнопки" новый текст
activeButton.textContent = 'Картинка №1';
```

- `innerText` — то же самое, что и `textContent`;
- `tagName` — строка с именем тега, с помощью которого создан текущий элемент, приведенным *к верхнему регистру*. Доступно только для чтения.

Класс `HTMLElement` поддерживает также следующие методы:

- `getBoundingClientRect()` — возвращает объект класса `DOMRect` с координатами и размерами текущего элемента, представленными в пикселах. Класс `DOMRect` поддерживает свойства:
  - `left` — горизонтальная координата левого верхнего угла элемента относительно левого верхнего угла видимой части его родителя;
  - `top` — вертикальная координата левого верхнего угла элемента относительно левого верхнего угла видимой части его родителя;
  - `right` — горизонтальная координата правого нижнего угла элемента относительно правого нижнего угла видимой части его родителя;
  - `bottom` — вертикальная координата правого нижнего угла элемента относительно правого нижнего угла видимой части его родителя;
  - `x` — то же самое, что и `left`;
  - `y` — то же самое, что и `top`;
  - `width` — ширина элемента с учетом внутренних просветов и рамки;
  - `height` — высота элемента с учетом внутренних просветов и рамки.

  Этот метод удобно использовать в элементе с прокруткой для выяснения, видим ли какой-либо из его потомков на экране. В *разд. 7.4* мы рассмотрим пример его использования;
- `scrollIntoView([<выравнивание>])` — прокручивает содержимое родителя текущего элемента таким образом, чтобы текущий элемент стал видимым на экране. Если в качестве *выравнивания* указать `true` или вообще не задавать его, текущий элемент будет выровнен по верхней границе родителя, если указать `false` — по нижней;
- `isEqualNode(<сравниваемый элемент>)` — возвращает `true`, если текущий и *сравниваемый* элементы полностью одинаковы (т. е. созданы одним и тем же тегом, имеют одинаковый набор атрибутов и одинаковое содержимое), и `false` в противном случае;
- `isSameNode(<сравниваемая ссылка>)` — возвращает `true`, если *сравниваемая ссылка* указывает на текущий элемент, и `false` в противном случае:

```
<nav id="buttons">
    <div id="1" class="active">1</div>
    <div id="2">2</div>
    <div id="3">3</div>
</nav>
. . .
const lastBtn1 = document.querySelector('nav div:last-child');
const lastBtn2 = document.getElementById('buttons').lastElementChild;
let result = lastBtn1.isSameNode(lastBtn2); // true
```

7.3. Упражнение. Делаем стильную полосу прокрутки

Элементы с прокруткой встречаются весьма часто. Чтобы оживить их, применяются стильные полосы прокрутки, реализуемые программно. Давайте и мы сделаем такой элемент и заодно закрепим на практике материал из *разд. 7.1* и *7.2*.

1. В папке 7\!sources сопровождающего книгу электронного архива (см. *приложение 3*) найдем страницу 7.1.html и скопируем ее куда-либо на локальный диск.

   Эта страница содержит элемент с прокруткой и расположенную правее заготовку для полосы прокрутки в виде вертикальной линии с овальным серым захватом (рис. 7.2).

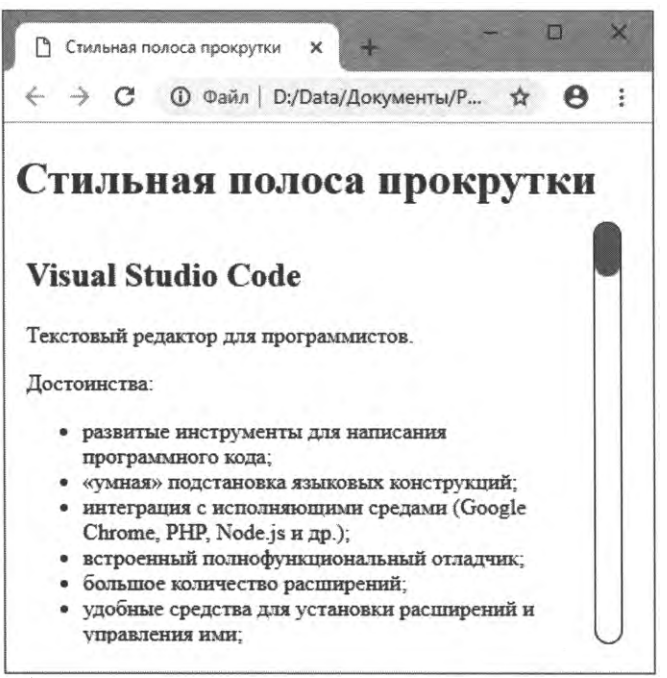


Рис. 7.2. Элемент с заготовкой для полосы прокрутки

2. Откроем страницу 7.1.html в текстовом редакторе.

   Оформление элементов задается внутренней таблицей стилей, имеющейся в коде страницы. Вот код, создающий элемент с прокруткой и полосу прокрутки:

```
<div id="container">
    <h2>Visual Studio Code</h2>
    . . .
</div>
<div id="scrollbar">
    <div></div>
</div>
```

Блок с якорем `container` создает элемент с прокруткой, а следующий за ним блок с якорем `scrollbar` — полосу прокрутки. Захват (движок), расположенный в полосе прокрутки, также создан блоком.

3. Запишем в конце кода страницы тег `<script>` и вставим в него первые выражения сценария, которые извлекут объекты, представляющие элемент с прокруткой, полосу прокрутки и ее захват:

```
<script type="text/javascript">
    const container = document.getElementById('container');
    const scrollbar = document.getElementById('scrollbar');
    const handle = scrollbar.firstElementChild;
</script>
```

Объект-захват мы можем извлечь, обратившись к свойству `firstElementChild` полосы прокрутки, поскольку он является ее первым (и единственным) потомком.

4. Чтобы вычислить величину, на которую нужно сместить захват и содержимое элемента `container`, нам понадобится хранить предыдущее положение курсора мыши. Объявим переменную, где она будет храниться:

```
let lastPos = 0;
```

5. Привяжем к захвату `handle` обработчик события `mousemove`, который запустит процесс перетаскивания захвата:

```
handle.addEventListener('mousedown', function(evt) {
    lastPos = evt.clientY;
    this.addEventListener('mousemove', mMove);
    this.addEventListener('mouseup', mUp);
});
```

В теле этого обработчика мы сохраняем текущую вертикальную координату курсора мыши, вычисленную относительно страницы, и привязываем к захвату обработчики событий `mousemove` и `mouseup`.

Обработчик события `mousemove` станет самым сложным. Он будет вычислять смещение курсора мыши относительно ее предыдущего положения и на его основе определять положение захвата и величину, на которую следует прокрутить содержимое элемента `container`.

6. Вставим перед предыдущим выражением код, объявляющий обработчик этого события:

```
function mMove(evt) {
    let d = evt.clientY - lastPos;
    let y = this.offsetTop + d, cy;
    if (y > 0 && y < 262) {
        this.style = 'top: ' + y + 'px;';
        cy = (container.scrollHeight - container.clientHeight) * y / 262;
        container.scrollTop = cy;
```

```
            lastPos = evt.clientY;
        }
    }
```

Здесь мы сначала вычисляем смещение курсора мыши (первое выражение тела) и на его основе получаем величину смещения захвата (второе выражение).

Далее проверяем, не выходит ли полученное смещение захвата за пределы полосы прокрутки. Для этого достаточно выяснить, укладывается ли она в диапазон `0...262`, где последнее число — это высота полосы прокрутки (300 пикселов) за вычетом высоты захвата (38 пикселов, обе величины записаны в стилях).

Если смещение укладывается в указанный диапазон, создаем встроенный стиль, указывающий вертикальную координату захвата, и присваиваем его свойству `style`. Потом вычисляем величину, на которую нужно прокрутить содержимое элемента `container`, и присваиваем ее свойству `scrollTop`. Напоследок не забудем сохранить текущее положение мыши по вертикали.

7. Впишем под предыдущим выражением объявление обработчика события `mouseup`, который прервет операцию перетаскивания:

```
function mUp(evt) {
    this.removeEventListener('mousemove', mMove);
    this.removeEventListener('mouseup', mUp);
}
```

Он просто убирает привязку обработчиков событий `mousemove` и `mouseup`.

Откроем страницу 7.1.html в веб-обозревателе и протащим захват полосы прокрутки вниз (рис. 7.3).

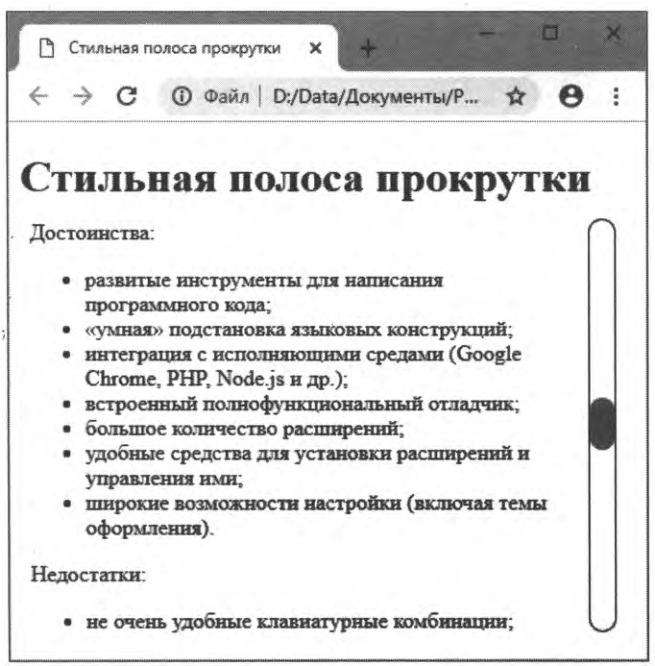


Рис. 7.3. Захват полосы прокрутки протащен вниз

7.4. Упражнение. Управляем стилями

Сделаем еще один элемент с прокруткой, в котором навигация на нужный фрагмент содержимого выполняется щелчком на гиперссылке в панели навигации, а сама гиперссылка при этом визуально выделяется.

1. В папке 7\!sources сопровождающего книгу электронного архива (см. *приложение 3*) отыщем страницу 7.2.html и скопируем ее в какое-либо место локального диска.

   Эта страница содержит элемент с прокруткой и панель навигации с гиперссылками (рис. 7.4). Для оформления элементов здесь также применяется внутренняя таблица стилей.

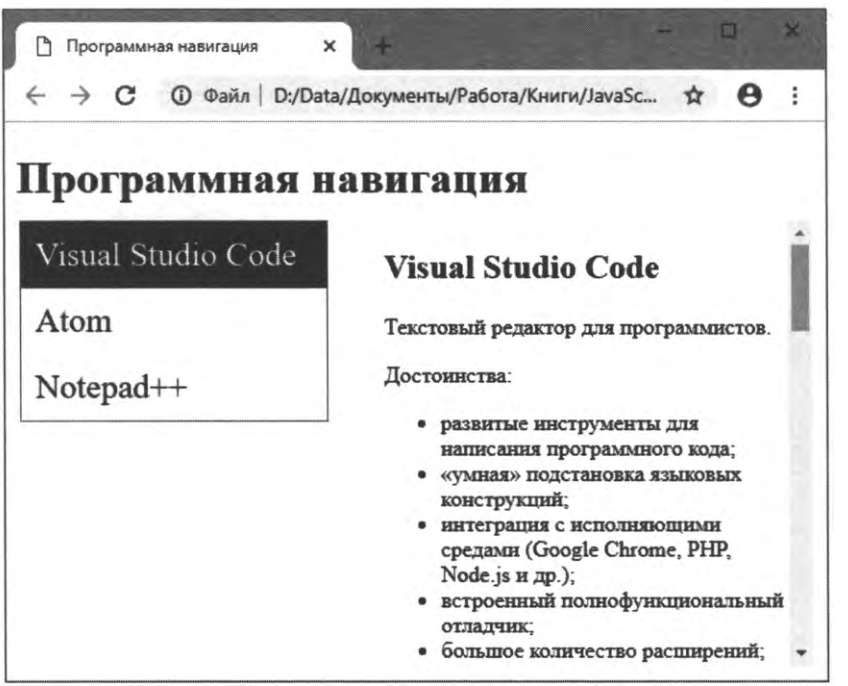


Рис. 7.4. Элемент с прокруткой и панель навигации с гиперссылками

2. Откроем код страницы 7.2.html в текстовом редакторе:

```
<nav>
    <a href="#vsc" class="jslink active">Visual Studio Code</a>
    <a href="#atom" class="jslink">Atom</a>
    <a href="#notepadpp" class="jslink">Notepad++</a>
</nav>
<section>
    <h2 id="vsc ">Visual Studio Code</h2>
    . . .
    <h2 id="atom">Atom</h2>
    . . .
```

```
    <h2 id="notepadpp">Notepad++</h2>
    . . .
</section>
```

Панель навигации формируется тегом <nav>. В ней находятся гиперссылки с привязанным стилевым классом jslink, которые ссылаются на якоря. Первая, активная, гиперссылка помечена стилевым классом active.

Сам элемент с прокруткой создан тегом <section>. Заголовки фрагментов, на которые должна выполняться навигация при щелчках на гиперссылках, помечены соответствующими якорями.

Навигация на странице уже работает: если щелкнуть на любой гиперссылке, содержимое элемента будет прокручено таким образом, чтобы соответствующий заголовок появился на экране. Все это реализуется веб-обозревателем.

От нас требуется сделать так, чтобы гиперссылка, на которой был выполнен щелчок, выделялась стилевым классом active.

3. В конце кода страницы добавим тег <script> с выражениями, получающими коллекцию гиперссылок, тег <section> и коллекцию заголовков второго уровня (<h2>), у которых заданы якоря:

```
const links = document.querySelectorAll('.jslink');
const container = document.getElementsByTagName('section')[0];
const anchors = document.querySelectorAll('h2[id]');
```

4. Объявим функцию-обработчик события click гиперссылок и сразу же выполним ее привязку:

```
function navigate() {
    let e = document.querySelector('nav .active');
    if (e)
        e.classList.remove('active');
    this.classList.add('active');
}
links.forEach(
    (el) => { el.addEventListener('click', navigate) }
);
```

В теле функции ищем гиперссылку, ранее помеченную как активная, и убираем у нее стилевой класс active, после чего привязываем его к гиперссылке, на которой был выполнен щелчок.

Свойство classList элемента страницы хранит объект-коллекцию всех привязанных к элементу стилевых классов. Метод add привязывает к элементу заданный стилевой класс, а метод remove удаляет стилевой класс из числа привязанных.

5. Привяжем обработчик к событию scroll тега <section>, который сделает активной гиперссылку, соответствующую заголовку второго уровня, который в текущий момент видим на экране:

```
container.addEventListener('scroll', (evt) => {
    let i, anchor, y, a;
    for (i = 0; i < anchors.length; ++i) {
        anchor = anchors[i];
        y = anchor.getBoundingClientRect().top;
        if (y > 0 && y < container.clientHeight) {
            links.forEach((l) => {
                if (l.href.endsWith(anchor.id))
                    l.classList.add('active');
                else
                    l.classList.remove('active');
            });
            break;
        }
    }
});
```

Мы просто перебираем все заголовки второго уровня, пока не найдем тот, чья вертикальная координата находится в диапазоне от `0` до высоты видимой части тега `<section>`. Далее перебираем гиперссылки, пока не найдем ссылающуюся на этот заголовок, и привязываем к ней стилевой класс `active`, а у всех остальных гиперссылок убираем его.

Откроем страницу 7.2.html в веб-обозревателе и проверим ее в работе.

7.4.1. Средства для управления стилевыми классами

Для указания стилевых классов у элемента можно использовать свойство `className`, соответствующее атрибуту тега `class`. Однако если к элементу уже привязаны какие-то стилевые классы, и нужно привязать еще один или, наоборот, убрать привязку какого-либо стилевого класса, пользоваться этим свойством неудобно.

Класс `HTMLElement` поддерживает доступное только для чтения свойство `classList`. Оно хранит объект класса `DOMTokenList` — коллекцию привязанных к элементу стилевых классов.

Класс `DOMTokenList` поддерживает методы:

- `add(`*<имена стилевых классов>*`)` — привязывает к текущему элементу стилевые классы с указанными в вызове *именами*. Если какой-либо из стилевых классов уже привязан, его привязка не выполняется:

```
l.classList.add('jslink', 'special', 'active');
```

- `remove(`*<имена стилевых классов>*`)` — убирает привязку к текущему элементу стилевых классов с заданными *именами*:

```
l.classList.remove('active');
```

- `toggle(`*<имя стилевого класса>*`)` — если стилевой класс с указанным *именем* не привязан к текущему элементу, выполняет его привязку, в противном случае — удаляет его;

- `contains(<имя стилевого класса>)` — возвращает `true`, если стилевой класс с указанным *именем* привязан к текущему элементу, и `false` в противном случае:

```
if (l.classList.contains('special'))
    . . .
```

7.4.2. Средства для управления встроенными стилями

Класс `HTMLElement` поддерживает свойство `style`, соответствующее одноименному атрибуту стиля. Этому свойству можно присвоить CSS-код, создающий встроенный стиль:

```
el.style = 'color: black; background-color: white;';
```

Но `style` — необычное свойство. Если извлечь его значение, будет получена не строка с CSS-кодом встроенного стиля, а объект класса `CSSStyleDeclaration`, представляющий этот стиль.

Класс `CSSStyleDeclaration` поддерживает ряд свойств, представляющих различные атрибуты стиля. Их имена формируются по следующим правилам:

- если имя атрибута стиля состоит из одного слова — свойства имеет то же имя;
- если имя атрибута стиля состоит из нескольких слов, разделенных дефисами, — имя свойства совпадает с именем атрибута стиля, в котором дефисы удалены, а следующие за ними буквы приведены к верхнему регистру.

Примеры именования свойств класса приведены в табл. 7.2.

Таблица 7.2. *Примеры именования свойств класса* `CSSStyleDeclaration`

Имя атрибута стиля	Имя свойства
color	color
background-color	backgroundColor
border-bottom-style	borderBottomStyle

А вот примеры их использования:

```
el.style.color = 'black';
el.style.backgroundColor = 'white';
```

7.5. Изменение содержимого элементов

Для изменения содержимого элементов страниц класс `HTMLElement` предлагает три свойства, доступных для чтения и записи:

- `innerHTML` — хранит HTML-код содержимого текущего элемента:

```
<p id="par1">JavaScript</p>
. . .
const par1 = document.getElementById('par1');
```

```
par1.innerHTML = 'Язык программирования <em>' + par1.innerHTML + '</em>';
/*
    Результат:
    Язык программирования JavaScript
*/
```

- `textContent` и `innerText` — хранят текстовое содержимое элемента:

```
<p id="par2">JavaScript</p>
. . .
const par2 = document.getElementById('par2');
par2.textContent = 'Язык программирования ' + par2.innerText;
/*
    Результат:
    Язык программирования JavaScript
*/
```

Если присвоить этим свойствам HTML-код, все содержащиеся в нем теги будут выведены как есть:

```
par2.textContent = 'Язык программирования <em>' + par2.innerText + '</em>';
/*
    Результат:
    Язык программирования <em>JavaScript</em>
*/
```

Класс `HTMLElement` поддерживает два полезных метода:

- `insertAdjacentHTML(`*<местоположение>*`, `*<HTML-код>*`)` — вставляет указанный *HTML-код* по заданному относительно текущего элемента *местоположению*. В качестве *местоположения* указывается одна из строк:
  - `'afterbegin'` — после открывающего тега текущего элемента (в начало его содержимого);
  - `'beforeend'` — перед закрывающим тегом текущего элемента (в конец содержимого);
  - `'beforebegin'` — перед открывающим тегом текущего элемента;
  - `'afterend'` — после закрывающего тега текущего элемента.

Пример:

```
<p>язык программирования<p>
. . .
const p1 = document.getElementsByTagName('p')[0];
const s1 = '<strong>JavaScript</strong> - ';
p1.insertAdjacentHTML('afterbegin', s1);
const s2 = '<p>Применяется в веб-разработке</p>';
p1.insertAdjacentHTML('afterend', s2);
```

```
/*
    Результат:
    JavaScript - язык программирования
    Применяется в веб-разработке
*/
```

- `insertAdjacentText(<местоположение>, <текст>)` — то же самое, что и `insertAdjacentHTML`, только вставляет обычный *текст*.

7.5.1. Методы *write* и *writeln*

Добавить новое содержимое в элементы страницы и даже создать новые элементы можно с помощью следующих методов класса `HTMLDocument`:

- `write(<выводимые значения>)` — знакомый нам метод, выводит указанные *значения* в том месте страницы, в котором находится его вызов. *Значения* выводятся вплотную друг к другу, без пробелов между ними:

```
// Создаем в этом месте новый абзац
document.write('<p>', 'HTML и CSS', '</p>');
```

- `writeln(<выводимые значения>)` — то же самое, что и `write`, но после всех значений выводит разрыв строк (последовательность символов `\r\n`). Разрыв строк отображается веб-обозревателем в виде обычного пробела.

Применение методов `write` и `writeln` — самый простой (но, к сожалению, самый медленный) способ вывести данные на экран. Он часто используется в простых сценариях и при обучении программированию.

> Если методы `write` и `writeln` используются в обработчике события, выведенные ими данные полностью заменят исходное содержание страницы. Поэтому в обработчиках событий применять их нельзя.

7.6. Упражнение. Создаем новые элементы веб-страниц путем конструирования

Возьмем упражнение, выполненное в *разд. 3.8* (вычисление длин окружностей диаметром 2, 5, 10 и 20 м), и перепишем его так, чтобы не использовать для вывода результатов «медленные» методы `write` и `writeln`. Мы применим другой, более производительный подход — конструирование элементов путем создания представляющих их объектов DOM и помещения их на страницу.

Элементы, созданные путем конструирования, обрабатываются и выводятся на экран быстрее элементов, созданных обращением к свойству и методам, описанным в *разд. 7.5*, поскольку веб-обозревателю не приходится разбирать HTML-код.

1. В папке 7\!sources сопровождающего книгу электронного архива (см. *приложение 3*) найдем исходные файлы: страницу 3.1.html и файл сценария 3.8.js. Скопируем их в какое-либо место на локальном диске.

2. Откроем страницу 3.1.html в текстовом редакторе и удалим из секции тела страницы тег <table> — в нем находится сценарий, выполняющий вычисления и вывод таблицы с результатами (удаленные фрагменты кода показаны здесь зачеркнутыми):

```
<body>
    <h1>Длина окружности</h1>
    <table>
        . . .
    </table>
</body>
```

3. Добавим в конце HTML-кода тег <script> со сценарием, создающим массив с диаметрами окружностей, у которых нужно вычислить длины, и переменную, которую мы используем в качестве счетчика цикла:

```
<script type="text/javascript">
    const ds = [2, 5, 10, 20];
    let i;
</script>
```

4. Добавим выражение, которое создаст объект, представляющий тег <table>:

```
const table = document.createElement('table');
```

Для создания объекта-тега применен метод createElement.

5. Добавим два выражения, создающие аналогичным образом объект-строку и объект-ячейку шапки:

```
let row = document.createElement('tr');
let cell = document.createElement('th');
```

6. Добавим выражение, которое создаст содержимое ячейки — текст «Диаметр»:

```
let text = document.createTextNode('Диаметр');
```

Для создания текстового содержимого мы применили здесь метод createTextNode.

7. Запишем выражение, которое добавит готовое текстовое содержимое в ячейку:

```
cell.appendChild(text);
```

Метод appendChild добавит указанный в его параметре элемент в тот, у которого был вызван.

8. Добавим готовую ячейку в строку:

```
row.appendChild(cell);
```

9. Создадим вторую ячейку строки и добавим готовую строку в таблицу:

```
cell = document.createElement('th');
text = document.createTextNode('Длина окружности');
cell.appendChild(text);
row.appendChild(cell);
table.appendChild(row);
```

10. В цикле переберем массив с величинами диаметров и в его теле создадим остальные строки таблицы:

```
for (i = 0; i < ds.length; ++i) {
    row = document.createElement('tr');
    cell = document.createElement('td');
    cell.textContent = ds[i];
    row.appendChild(cell);
    cell = document.createElement('td');
    cell.textContent = circleLength(ds[i]);
    row.appendChild(cell);
    table.appendChild(row);
}
```

Здесь для создания текстового содержимого ячеек мы использовали свойство `textContent`, а не метод `createTextNode`. Обращение к свойству выполняется несколько медленнее, чем вызов метода, но позволяет немного сократить код.

11. И наконец, добавим таблицу в тело страницы, чтобы вывести ее на экран:

```
document.body.appendChild(table);
```

Если мы не сделаем этого, с таким трудом сконструированная нами таблица так и останется лишь в памяти компьютера.

Откроем переделанную страницу 3.1.html в веб-обозревателе и посмотрим на таблицу с результатами (рис. 7.5).

Рис. 7.5. Сконструированная нами таблица в окне веб-обозревателя

7.6.1. Методы, конструирующие элементы веб-страниц

Для создания элементов страниц применяются два метода класса `HTMLDocument`, представляющего саму страницу:

- `createElement(<`*имя тега*`>)` — создает объект, представляющий тег с указанным *именем*, и возвращает его в качестве результата. *Имя тега* записывается в виде строки без символов < и >;
- `createTextNode(<`*текст*`>)` — создает объект класса `Text`, представляющий текстовое содержимое элемента, на основе указанного *текста* и возвращает его в качестве результата.

Вывод сконструированного элемента выполняется добавлением его в другой элемент в качестве потомка. Это выполняют следующие методы класса `HTMLElement`:

- `appendChild(<элемент>)` — добавляет указанный *элемент* — тег или текстовое содержимое — в конец текущего, после чего возвращает добавленный элемент в качестве результата;
- `insertBefore(<вставляемый элемент>, <элемент, перед которым его следует поместить>)` — вставляет указанный в первом параметре элемент *перед* элементом, заданном во втором параметре. Если вторым параметром передать `null`, *добавляемый элемент* будет помещен в конце текущего элемента (как при вызове метода `appendChild`). Вставленный элемент возвращается как результат:

```
<p>Первый абзац</p>
<p>Второй абзац</p>
. . .
const ps = document.getElementsByTagName('p');
const par3 = document.createElement('p');
par3.textContent = 'Третий абзац';
document.body.insertBefore(par3, ps[1]);
/*
    Результат:
    Первый абзац
    Третий абзац
    Второй абзац
*/
```

- `insertAdjacentElement(<местоположение>, <вставляемый элемент>)` — вставляет указанный *элемент* по указанному относительно текущего элемента *местоположению*. В качестве *местоположения* задается одна из строк:
  - `'afterbegin'` — после открывающего тега текущего элемента (в результате *вставляемый элемент* станет его первым потомком);
  - `'beforeend'` — перед закрывающим тегом текущего элемента (*вставляемый элемент* станет его последним потомком);
  - `'beforebegin'` — перед открывающим тегом текущего элемента (*вставляемый элемент* станет его предыдущим соседом);
  - `'afterend'` — после закрывающего тега текущего элемента (*вставляемый элемент* станет его следующим соседом).

  Пример:

```
<p> - язык программирования<p>
. . .
const p2 = document.getElementsByTagName('p')[0];
const strong = document.createElement('strong');
strong.textContent = 'JavaScript';
p2.insertAdjacentElement('afterbegin', strong);
```

```
const par4 = document.createElement('p');
par4.textContent = 'Применяется в веб-разработке';
p2.insertAdjacentElement('afterend', par4);
/*
    Результат:
    JavaScript - язык программирования
    Применяется в веб-разработке
*/
```

Еще два метода пригодятся, если нужно заменить один элемент другим или вообще удалить его:

- `replaceChild(<заменяющий элемент>, <заменяемый элемент>)` — заменяет один элемент другим и возвращает в качестве результата *заменяемый элемент*. Вызывается у родителя *заменяемого элемента*:

```
<p>HTML</p>
<p>Язык программирования</p>
. . .
const p3 = document.getElementsByTagName('p')[0];
const h1 = document.createElement('h1');
h1.textContent = 'Python';
document.body.replaceChild(h1, p3);
/*
    Результат:
    Python
    Язык программирования
*/
```

- `removeChild(<удаляемый элемент>)` — удаляет указанный *элемент* и возвращает его в качестве результата. Вызывается у родителя *удаляемого элемента*:

```
<ul>
    <li>JavaScript</li>
    <li>PHP</li>
    <li>Delphi</li>
</ul
. . .
const ul = document.getElementsByTagName('ul')[0];
document.body.removeChild(ul.lastElementChild);
/*
    Результат:
    JavaScript
    PHP
*/
```

7.7. Получение сведений о веб-странице

Класс `HTMLDocument`, представляющий страницу, поддерживает ряд свойств, хранящих сведения о самой странице:

- `title` — название текущей страницы, заданное в теге `<title>`. Это единственное из упомянутых здесь свойств, доступно для записи (все остальные свойства доступны только для чтения):

```
// Указываем для страницы новое название
document.title = 'Практикум JavaScript';
```

- `URL` — полный интернет-адрес страницы в виде строки;
- `domain` — адрес хоста, извлеченный из полного интернет-адреса, в виде строки;
- `referrer` — интернет-адрес страницы, с которой был выполнен переход на текущую страницу, в виде строки. Если текущая страница была открыта не в результате перехода (а, например, набором интернет-адреса вручную или через вкладки веб-обозревателя), хранит пустую строку;
- `characterSet` — кодировка, заданная для страницы, в виде строки;
- `lastModified` — дата и время последнего изменения страницы в виде строки формата:

<месяц>/<число>/<год> <часы>:<минуты>:<секунды>.

7.8. Самостоятельные упражнения

- Напишите страницу 7.8.1.html со сценарием, который выводит на экран обозначения нажатых клавиш и их коды в десятичной и шестнадцатеричной системах счисления.

  Подсказка: обрабатывайте событие `keydown` в секции тела страницы, а для вывода числа в шестнадцатеричной системе используйте метод `toString` класса `Number`.

- Напишите страницу 7.8.2.html со свободно позиционируемым элементом, который можно перемещать мышью. Эта страница может выглядеть, как показано на рис. 7.6 (перемещаемый элемент закрашен серым фоном).

- В папке 7\!sources сопровождающего книгу электронного архива (см. *приложение 3*) найдите файл сценария 7.8.js. Он содержит код, создающий служебный объект класса `Object` со сведениями о текстовом редакторе Atom (приведен с сокращениями):

```
const atom = {
    name: 'Atom',
    desc: 'Текстовый редактор для программистов.',
    pros: [
        'поддержка практически всех языков . . .',
        . . .
    ],
```

```
    contras: [
        'некоторые полезные инструменты . . .',
        . . .
    ],
    site: 'https://atom.io/'
};
```

Напишите страницу 7.8.3.html со сценарием, формирующим на основе этого объекта описание текстового редактора (рис. 7.7).

Подсказка: сконструируйте все необходимые элементы.

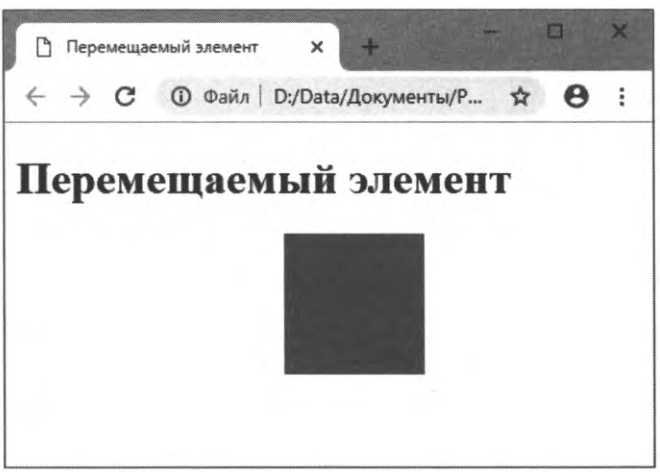


Рис. 7.6. Страница со свободно позиционируемым элементом, который можно перемещать мышью

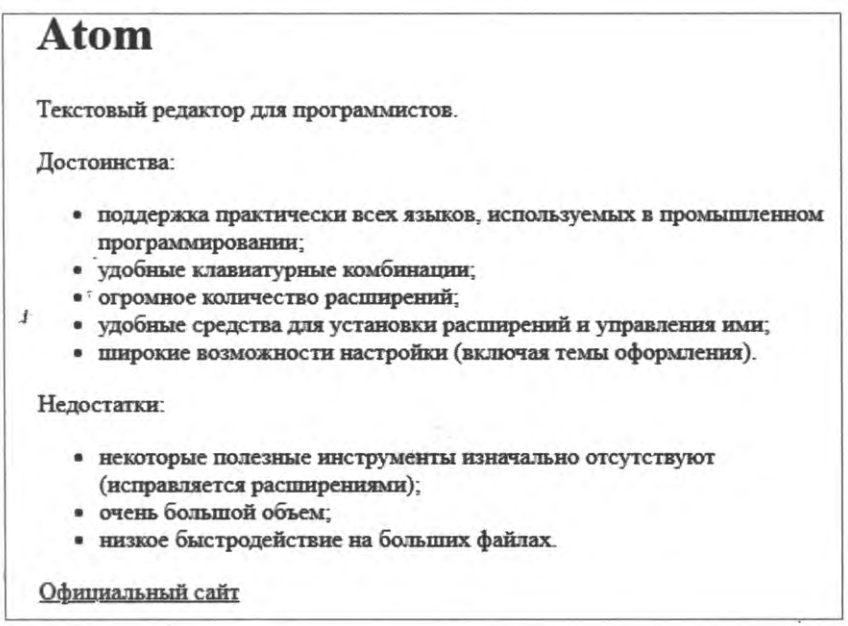


Рис. 7.7. Веб-страница с описанием текстового редактора

Урок 8
Графика и мультимедиа

Получение сведений о графических изображениях
Управление аудио- и видеороликами
Программный видеопроигрыватель

8.1. Получение сведений о графических изображениях

Мы можем получить некоторые сведения о графическом изображении, выведенном в теге `<img>`, — в частности, его изначальные размеры (разрешение). Для этого применяются специфические свойства класса `HTMLImageElement`, производного от класса `HTMLElement` и представляющего этот тег. Все эти свойства доступны только для чтения:

- `naturalWidth` — изначальная ширина изображения в виде числа в пикселах;
- `naturalHeight` — изначальная высота изображения в виде числа в пикселах;
- `complete` — `true`, если загрузка файла с изображением завершена, и `false` в противном случае.

Также поддерживаются два дополнительных события, возникающие:

- `load` — по окончании загрузки файла с изображением:

```
<img src="image.png" id="img1">
. . .
const img1 = document.getElementById('img1');
img1.addEventListener('load', () => {
    // Получаем изначальные размеры изображения
    let w = img1.naturalWidth;
    let h = img1.naturalHeight;
});
```

- `error` — при невозможности загрузить файл с изображением (например, файл по указанной в теге `<img>` ссылке не существует).

Оба события невсплывающие (см. *разд. 6.4*), не имеют обработчика по умолчанию (см. *разд. 6.5*) и представляются объектом класса `Event`.

8.2. Управление мультимедийными элементами

Мы можем получать сведения об аудио- и видеороликах, выведенных в тегах <audio> и <video>, — в частности, продолжительность и разрешение (у видеороликов). Мы также можем управлять их воспроизведением: запускать, приостанавливать, изменять текущую позицию, настраивать уровень громкости, приглушать и вновь восстанавливать звук и др.

Для этого применяются специфические свойства, методы и события классов HTMLAudioElement и HTMLVideoElement, производных от класса HTMLElement и представляющих теги <audio> и <video>. Рассмотрим их подробно.

8.2.1. Свойства

Классы HTMLAudioElement и HTMLVideoElement поддерживают следующие свойства:

- currentTime — текущая позиция воспроизведения ролика в виде числа в секундах:

```
// Устанавливаем позицию воспроизведения видеоролика на 10 секунд
video1.currentTime = 10;
```

- volume — громкость звукового сопровождения в виде числа с плавающей точкой от 0.0 (звук вообще не слышен) до 1.0 (максимальная громкость);
- muted — true, если звуковое сопровождение при текущем воспроизведении ролика приглушено, false в противном случае;
- playbackRate — относительная скорость текущего воспроизведения ролика. Указывается в виде числа с плавающей точкой: число меньше 1 задает замедленное воспроизведение, больше 1 — ускоренное, отрицательное число — воспроизведение в обратном направлении. Значение по умолчанию — 1.0 (воспроизведение с обычной скоростью):

```
// Замедленное вдвое воспроизведение
video1.playbackRate = 0.5;
// Ускоренное втрое воспроизведение
video1.playbackRate = 3.0;
// Воспроизведение в обратном направлении с нормальной скоростью
video1.playbackRate = -1.0;
```

- duration — продолжительность ролика в виде числа в секундах (только для чтения);
- videoWidth — ширина видеоролика в пикселах (только для чтения и только у видеороликов);
- videoHeight — высота видеоролика в пикселах (только для чтения и только у видеороликов);

- `paused` — `true`, если воспроизведение ролика приостановлено, `false` в противном случае (только для чтения);
- `seeking` — `true`, если пользователь перемещает регулятор текущей позиции ролика, `false` в противном случае (только для чтения);
- `ended` — `true`, если воспроизведение ролика закончилось, `false` в противном случае (только для чтения);
- `defaultMuted` — то же, что и `muted`, но задает состояние звука по умолчанию, которое будет использовано при последующих воспроизведениях ролика;
- `defaultPlaybackRate` — то же, что и `playbackRate`, но задает скорость воспроизведения по умолчанию, которая будет использована при последующих воспроизведениях ролика;
- `currentSrc` — интернет-адрес воспроизводимого в текущий момент файла. Может отличаться от заданного в атрибуте тега `src` или в свойстве `src`, если веб-сервер, с которого запрашивается файл, выполнил перенаправление. Доступно только для чтения;
- `networkState` — статус сетевой активности в виде числа:
  - `0` — файл еще не загружался;
  - `1` — файл загружен;
  - `2` — файл загружается;
  - `3` — файл не найден.

  Доступно только для чтения;
- `readyState` — статус самого ролика в виде числа:
  - `0` — сведения о ролике недоступны;
  - `1` — загружены только метаданные ролика (в частности, его разрешение и продолжительность);
  - `2` — загружен текущий кадр ролика;
  - `3` — загружены текущий и следующий кадры ролика;
  - `4` — загружено достаточно данных для бесперебойного воспроизведения ролика.

  Доступно только для чтения.

Помимо этого, поддерживаются свойства, соответствующие атрибутам тегов. Так свойство `src`, соответствующее атрибуту тега `src`, хранит ссылку на воспроизводимый файл.

8.2.2. Методы

Классам `HTMLAudioElement` и `HTMLVideoElement` свойственны следующие методы:

- `load()` — перезагружает текущий ролик. Обычно применяется, чтобы остановить ролик и перемотать его в начало;

- `pause()` — приостанавливает ролик;
- `play()` — запускает или возобновляет воспроизведение ролика.

8.2.3. События

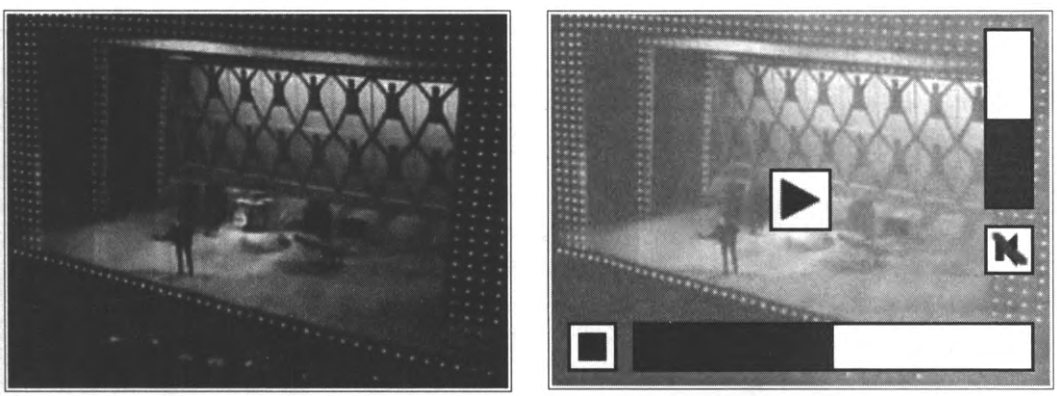
Все события, поддерживаемые мультимедийными элементами, невсплывающие, не имеют обработчика по умолчанию и представляются классом `Event`:

- `loadstart` — начата загрузка файла с роликом;
- `durationchange` — загружена часть метаданных, в которой записана продолжительность ролика;
- `loadedmetadata` — загружена остальная часть метаданных, в которой записано, в частности, разрешение ролика (если это видео);
- `loadeddata` — загружено достаточно данных, чтобы, по крайней мере, вывести первый кадр ролика, однако его воспроизведение будет вестись с паузами на подгрузку;
- `progress` — периодически возникает при загрузке файла;
- `canplay` — загружено достаточно данных, чтобы, по крайней мере, начать воспроизведение ролика, однако в дальнейшем возможны паузы на подгрузку;
- `canplaythrough` — загружено достаточно данных, чтобы начать воспроизведение ролика без пауз на подгрузку;
- `play` — воспроизведение ролика запущено или возобновлено нажатием кнопки пуска или вызовом метода `play`;
- `playing` — после события `play`, когда ролик реально начинает воспроизводиться. Между этими событиями возможна задержка, пока веб-обозреватель подгружает данные из файла;
- `timeupdate` — позиция воспроизведения ролика изменилась;
- `volumechange` — либо изменился уровень громкости, либо звук был приглушен, либо, наоборот, восстановлен;
- `waiting` — в воспроизведении ролика возникла пауза, необходимая для подгрузки;
- `pause` — воспроизведение ролика приостановлено;
- `seeking` — пользователь начал перемещать регулятор позиции воспроизведения;
- `seeked` — пользователь закончил перемещать регулятор позиции воспроизведения;
- `ratechange` — скорость воспроизведения ролика изменилась;
- `ended` — воспроизведение ролика закончено;
- `suspend` — загрузка мультимедийного файла либо полностью завершилась, либо приостановлена по какой-то причине (например, сервер перегружен);

- `stalled` — файл по какой-то причине не загружается (например, отсутствует на сервере);
- `abort` — прерывание загрузки файла вследствие сетевой неполадки.

8.3. Упражнение. Реализуем свой видеопроигрыватель

Создадим средствами HTML, CSS и JavaScript стильный видеопроигрыватель, в котором все элементы управления воспроизведением изначально скрыты (рис. 8.1, *а*) и появляются поверх видео при наведении на него курсора мыши (рис. 8.1, *б*).



а *б*

Рис. 8.1. Стильный видеопроигрыватель: *а* — все элементы управления воспроизведением скрыты, *б* — элементы управления появляются поверх видео при наведении на него курсора мыши

1. В папке 8\!sources сопровождающего книгу электронного архива (см. *приложение 3*) найдем страницу 8.1.html, а также папки: images (в которой хранятся изображения для кнопок) и media (с видеофайлом, который будет воспроизводиться в проигрывателе). Скопируем их в какое-либо место на локальном диске.
2. Откроем страницу 8.1.html в текстовом редакторе и найдем в ней код, создающий проигрыватель:

```
<div id="player">
    <video src="media/video.mp4"></video>
    <div class="interface">
        <div class="play"></div>
        <div class="stop"></div>
        <div class="position"><div></div></div>
        <div class="mute"></div>
        <div class="volume"><div></div></div>
    </div>
</div>
```

Он создается блоком с якорем `player`, в который вложены тег `<video>` и блок со стилевым классом `interface`, в котором и находятся все элементы управления воспроизведением. Они также создаются блоками, поскольку блоки проще оформить. В число элементов входят кнопка пуска и паузы (стилевой класс `play`), кнопка остановки (`stop`), регулятор позиции воспроизведения (`position`), кнопка приглушения звука (`mute`) и регулятор громкости (`volume`).

Блок со стилевым классом `interface` изначально скрыт и выводится при наведении на блок `player` курсора мыши. Его можно вывести принудительно, привязав к блоку `player` стилевой класс `active`.

В регуляторах указатели текущего положения создаются вложенными в них пустыми блоками. Задавая для такого блока ширину или высоту, мы можем указать текущее положение регулятора.

Изображения в кнопках выводятся в виде графического фона — так их проще позиционировать.

3. Добавим в конце HTML-кода тег `<script>` с выражениями, получающими доступ к нужным элементам страницы:

```
<script type="text/javascript">
    const player = document.getElementById('player');
    const video = player.firstElementChild;
    const play = player.querySelector('.play');
    const stop = player.querySelector('.stop');
    const position = player.querySelector('.position');
    const posH = position.firstElementChild;
    video.volume = 0.5;
</script>
```

Поскольку регулятор громкости изначально установлен на половину, мы сразу же задаем для тега `<video>` половинную громкость.

4. После загрузки достаточного для вывода первого кадра объема данных регулятор `position` следует установить в начало, а в кнопке `play` вывести изображение «пуск» из файла images\play.png. Также нужно вывести на экран блок с элементами управления, чтобы посетитель сайта сразу смог запустить видео.

Напишем обработчик события `canplay` тега `<video>`, который выполнит все это:

```
video.addEventListener('canplay', () => {
    posH.style.width = '0%';
    play.style.backgroundImage = 'url(images/play.png)';
    player.classList.add('active');
});
```

5. Если ролик в текущий момент воспроизводится, в кнопке `play` следует вывести изображение «пауза» (файл images\pause.png), а если поставлен на паузу — изображение «пуск». При этом нужно также, соответственно, скрывать и выводить блок с элементами управления.

Напишем обработчики событий `playing` и `pause` тега `<video>`, выполняющие эти действия:

```
video.addEventListener('playing', () => {
    play.style.backgroundImage = 'url(images/pause.png)';
    player.classList.remove('active');
});
video.addEventListener('pause', () => {
    play.style.backgroundImage = 'url(images/play.png)';
    player.classList.add('active');
});
```

6. В процессе воспроизведения ролика нужно обновлять текущую позицию в регуляторе `position`.

   Объявим обработчик события `timeupdate` тега `<video>` — он и сделает все, что нужно:

```
video.addEventListener('timeupdate', () => {
    posH.style.width = video.currentTime / video.duration * 100 + '%';
});
```

   Вычислить ширину указателя (пустого блока, находящегося в блоке-регуляторе) просто: мы делим текущую позицию воспроизведения на продолжительность ролика, умножаем полученное частное на 100 — и получаем ширину в процентах.

7. По окончании ролика следует привести проигрыватель в исходное положение. Проще всего сделать это, выполнив перезагрузку ролика — в обработчике события `ended` проигрывателя:

```
video.addEventListener('ended', () => {
    video.load();
});
```

8. Напишем обработчик события `click` кнопки `play`, который запустит воспроизведение ролика или, если он уже проигрывается, приостановит его:

```
play.addEventListener('click', () => {
    if (video.paused) {
        video.play();
    } else {
        video.pause();
    }
});
```

9. Обработчик события `click` кнопки `stop` приведет проигрыватель в исходное положение:

```
stop.addEventListener('click', () => {
    video.load();
});
```

10. Осталось сделать так, чтобы при щелчках на регуляторе `position` соответственно изменялась текущая позиция воспроизведения ролика.

Этого можно достичь, обрабатывая событие `click` регулятора `position`:

```
position.addEventListener('click', (evt) => {
    video.currentTime = evt.offsetX /
        position.clientWidth * video.duration;
});
```

Текущую позицию воспроизведения мы здесь вычисляем, разделив горизонтальную координату точки регулятора, на которой был выполнен щелчок, на его ширину и умножив полученное частное на продолжительность ролика.

Откроем страницу 8.1.html в веб-обозревателе, запустим ролик на воспроизведение, поставим на паузу и запустим вновь. Попробуем переместить регулятор позиции воспроизведения и посмотрим, что получится.

8.4. Самостоятельные упражнения

- В видеопроигрыватель на странице 8.1.html добавьте настройку уровня громкости регулятором `volume`.
- Там же реализуйте приглушение звука кнопкой `mute`. При этом, если звук приглушен, выведите на кнопке изображение из файла images\unmute.png, а если не приглушен — изображение из файла images\mute.png.

Урок 9
Веб-формы и элементы управления

Взаимодействие с элементами управления
Взаимодействие с веб-формами
Программная валидация

9.1. Взаимодействие с элементами управления

Мы можем программно получать значения, занесенные в элементы управления, временно делать их недоступными, реагировать на изменение значения в поле ввода, установку или сброс флажка и др. Все это выполняется посредством специфических свойств (табл. 9.1), методов (табл. 9.2) и событий (табл. 9.3), поддерживаемых классами, представляющими элементы управления (помимо полученных в «наследство» от суперкласса `HTMLElement`). Классы событий и их свойства представлены в табл. 9.4.

Таблица 9.1. *Свойства, поддерживаемые классами, представляющими элементы управления*

Свойство	Описание	Поддерживающие его элементы управления
`form`	Объект формы, в которой находится текущий элемент управления	Все
`value`	Занесенное значение в виде строки	Поля ввода, выбора и область редактирования
`defaultValue`	Изначальное значение, заданное в атрибуте тега `value`	
`selectionStart`	Номер первого символа выделенного фрагмента значения, если ничего не выделено — номер символа под текстовым курсором	
`selectionEnd`	Номер последнего символа выделенного фрагмента значения, если ничего не выделено — номер символа под текстовым курсором	
`autocomplete`	Если `'on'` — выводится список ранее введенных значений для быстрого выбора (поведение по умолчанию). Если `'off'` — такой список не выводится	

Таблица 9.1 *(окончание)*

Свойство	Описание	Поддерживающие его элементы управления
`checked`	Если `true` — флажок или переключатель установлен, если `false` — сброшен	Флажок и переключатель
`defaultChecked`	Соответствует атрибуту тега `checked`	
`files`	Объект-коллекция выбранных файлов (будет рассмотрен на *уроке 13*)	Поле выбора файлов
`options`	Объект-коллекция пунктов, имеющихся в списке	Список
`length`	Количество пунктов в списке	
`selectedIndex`	Номер выбранного пункта (нумерация начинается с нуля). Если ни один пункт не выбран, хранит значение `-1`. Если список позволяет выбирать несколько пунктов одновременно, хранит номер первого выбранного пункта	
`selected`	Если `true` — текущий пункт выбран, если `false` — не выбран	Пункт списка
`defaultSelected`	Соответствует атрибуту тега `selected`	
`index`	Порядковый номер текущего пункта	
`text`	Текст текущего пункта	

Таблица 9.2. *Методы, поддерживаемые классами, представляющими элементы управления*

Метод	Описание	Поддерживающие его элементы управления
`focus()`	Устанавливает фокус ввода на текущий элемент	Все
`blur()`	Удаляет фокус ввода с текущего элемента	
`select()`	Выделяет весь текст в области редактирования	Область редактирования
`stepUp([<d>])`	Увеличивает занесенное число на величину *d*. Если параметр не указан, выполняет инкремент	Поле ввода числа и регулятор
`stepDown([<d>])`	Уменьшает занесенное число на величину *d*. Если параметр не указан, выполняет декремент	
`add(<it>[, <p>])`	Добавляет новый пункт *it* в позицию *p*. Если позиция не указана, добавляет элемент в конец списка	Список
`remove(<p>)`	Удаляет пункт под номером *p*	

Таблица 9.3. *События, поддерживаемые классами, представляющими элементы управления*

Имя события	Когда возникает?
События, имеющие отношение к занесенному значению Всплывающие. Обработчик по умолчанию отсутствует	
`change`	Во флажках, переключателях и списках — при изменении состояния (установке, сбросе, выборе пункта списка). В полях ввода и областях редактирования — при потере фокуса, если занесенное значение было изменено
`input`	В полях ввода и областях редактирования — сразу при изменении занесенного значения. Во флажках, переключателях и списках не возникает
`invalid`	Занесенное в поле ввода или область редактирования значение некорректно. Возникает перед отправкой данных, выполняемой формой, в результате чего отправка отменяется
`select`	В полях ввода и областях редактирования — при выделении фрагмента значения
События фокуса ввода Обработчик по умолчанию отсутствует	
`focus`	Получение текущим элементом фокуса ввода (невсплывающее)
`focusin`	Получение текущим элементом фокуса ввода (всплывающее)
`blur`	Потеря текущим элементом фокуса ввода (невсплывающее)
`focusout`	Потеря текущим элементом фокуса ввода (всплывающее)
События буфера обмена Всплывающие	
`cut`	Вырезание выделенного текста (обработчик по умолчанию выполняет вырезание текста)
`paste`	Вставка текста из буфера обмена (обработчик по умолчанию выполняет вставку текста)

Таблица 9.4. *Классы событий и их свойства*

Свойство	Хранимое значение
Класс `FocusEvent` Производный от класса `Event`. Представляет события фокуса ввода	
`relatedTarget`	В случае события `focus` или `focusin` — элемент управления, потерявший фокус ввода. В случае события `blur` или `focusout` — элемент, получивший фокус.
Остальные события представляются классом `Event`	

9.1.1. Получение и обработка введенного в элемент значения

Получение и обработка введенного в элемент значения осуществляются в следующем порядке:

- обычные поля ввода, поля ввода пароля, интернет-адреса, адреса электронной почты, области редактирования — непосредственно из свойства `value`:

```
<input type="text" id="val">
. . .
const val = document.getElementById('val');
let value = val.value;
```

- поля ввода числа и регуляторы — полученное из свойства `value` значение, представляющее собой строку, следует преобразовать в числовой тип:

```
<input type="number" id="num">
. . .
const num = document.getElementById('num');
// Преобразуем значение в числовой тип
let number = parseInt(num.value);
// Также можно положиться на веб-обозреватель — он сам выполняет
// преобразование в ряде случаев (подробности — в разд. 1.11)
let a = num.value / 100;
```

- поля ввода даты — дата в свойстве `value` хранится в виде строки формата *<год>-<номер месяца>-<число>*:

```
<input type="date" id="dat">
. . .
const dat = document.getElementById('dat');
// Создаем на основе введенной даты объект класса Date
let d = new Date(dat.value);
```

- поля ввода времени — время в свойстве `value` хранится в виде строки формата *<часы>:<минуты>:<секунды>*;
- поля выбора цвета — цвет в свойстве `value` хранится в виде строки формата *#<R><G><B>*;
- флажки и переключатели — через свойство `checked`:

```
<input type="checkbox" id="chk">
. . .
const chk = document.getElementById('chk');
if (chk.checked)
    // Флажок установлен
else
    // Флажок сброшен
```

- список с возможностью выбора только одного пункта — номер выбранного пункта можно получить из свойства `selectedIndex`:

```
<select size="5" id="sel">
    <option value="html">HTML</option>
    <option value="css">CSS</option>
    <option value="js">JavaScript</option>
    <option value="php">PHP</option>
    <option value="python">Python</option>
</select>
. . .
const sel = document.getElementById('sel');
let selected = sel.selectedIndex;
```

- **список с возможностью выбора нескольких пунктов** — следует перебрать все пункты списка, коллекция которых доступна из свойства `options`, и выяснить, какие из них выбраны, проверкой свойства `selected`:

```
<select size="5" id="msel" multiple>
    <option value="html">HTML</option>
    <option value="css">CSS</option>
    <option value="js">JavaScript</option>
    <option value="php">PHP</option>
    <option value="python">Python</option>
</select>
. . .
const msel = document.getElementById('msel');
let selected = [];
// Помещаем значения (задается в атрибуте value тега <option>)
// выбранных пунктов в массив selected
for (let i = 0; i < sel.length; ++i)
    if (sel.options[i].selected)
        selected.push(sel.options[i].value);
```

9.1.2. Программное создание и удаление пунктов списка

Программное создание и удаление пунктов списка осуществляются в следующем порядке:

```
<select size="5" id="psel">
</select>
. . .
const psel = document.getElementById('psel');
// Создаем пункты в списке psel
let languageValues = ['html', 'css', 'js', 'php', 'python'];
let languageTexts = ['HTML', 'CSS', 'JavaScript', 'PHP', 'Python'];
let i, opt;
for (i = 0; i < languageValues.length; ++i) {
    opt = document.createElement('option');
    opt.value = languageValues[i];
```

```
    opt.textContent = languageTexts[i];
    psel.add(opt);
}
. . .
// Удаляем пункт JavaScript
psel.remove(2);
```

9.2. Упражнение. Пишем первое клиентское веб-приложение

Напишем клиентское веб-приложение, генерирующее CSS-код для создания у элемента страницы рамки на основе введенных пользователем данных.

> *Клиентское веб-приложение* — программа, реализованная в виде веб-страницы с веб-сценарием и не требующая для работы других программ (в частности, серверных веб-приложений, работающих на стороне веб-сервера).

1. В папке 9\!sources сопровождающего книгу электронного архива (см. *приложение 3*) найдем страницу 9.1.html, скопируем ее в какое-либо место на локальном диске и откроем в текстовом редакторе.

   Эта страница содержит набор элементов управления, в которых пользователь будет заносить необходимые сведения (рис. 9.1).

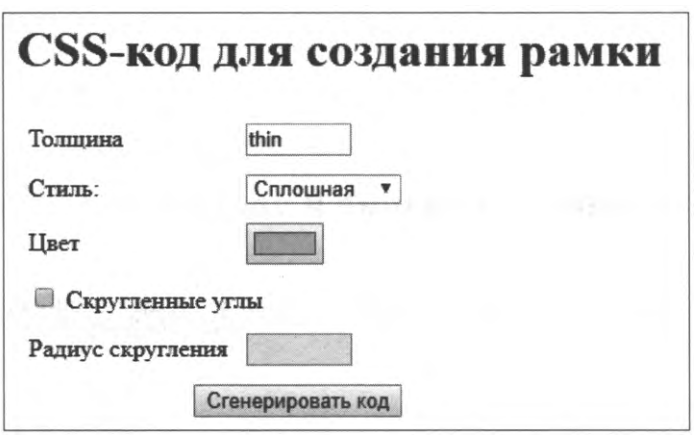


Рис. 9.1. Веб-страница с набором элементов управления

Вот перечень всех элементов страницы, которыми мы будем манипулировать программно, и заданных у них якорей:

- поле ввода **Толщина** — `width`;
- список **Стиль** — `style`;
- поле выбора цвета **Цвет** — `color`;

- флажок **Скругленные углы** — `rounded`;
- поле ввода **Радиус скругления** — `radius`;
- кнопка **Сгенерировать код** — `generate`;
- текст фиксированного форматирования (тег `<pre>`), в котором будет выводиться готовый CSS-код (он находится ниже кнопки, изначально пуст и поэтому не виден на экране) — `output`.

> Если значения, заносимые в элементы управления, не предназначены для пересылки на сервер, а будут полностью обрабатываться в веб-сценарии, элементы управления можно не заключать в веб-форму (в тег `<form>`).

2. После HTML-кода впишем тег `<script>` с выражениями, которые получат доступ к приведенным ранее элементам страницы:

```
<script type="text/javascript">
    const width = document.getElementById('width');
    const style = document.getElementById('style');
    const color = document.getElementById('color');
    const rounded = document.getElementById('rounded');
    const radius = document.getElementById('radius');
    const generate = document.getElementById('generate');
    const output = document.getElementById('output');
</script>
```

3. Поле ввода **Радиус скругления** должно быть доступно для ввода только тогда, когда установлен флажок **Скругленные углы**. Поэтому напишем обработчик события `change` флажка **Радиус скругления**, который будет делать поле ввода **Скругленные углы** доступным или недоступным, в зависимости от состояния флажка:

```
rounded.addEventListener('change', () => {
    radius.disabled = !rounded.checked;
});
```

4. Напишем код, который при нажатии кнопки **Сгенерировать код** сгенерирует и выведет на экран CSS-код для задания рамки:

```
generate.addEventListener('click', () => {
    let s = 'border: ' + width.value + ' ' +
        style.options[style.selectedIndex].value + ' ' + color.value +
        ';\r\n';
    if (rounded.checked)
        s += 'border-radius: ' + radius.value + ';';
    output.textContent = s;
});
```

В случае задания скругленных углов сгенерированный CSS-код будет состоять из двух атрибутов стиля: `border` и `border-radius`, — выведенных в две строки. Мы добились этого, поместив после символа `;` первой строки последователь-

ность литералов \r\n, создающих разрыв строки. Текст фиксированного форматирования (тег <pre>) выведет этот разрыв как есть.

Откроем страницу 9.1.html в веб-обозревателе и проверим наше первое клиентское веб-приложение в действии (рис. 9.2).

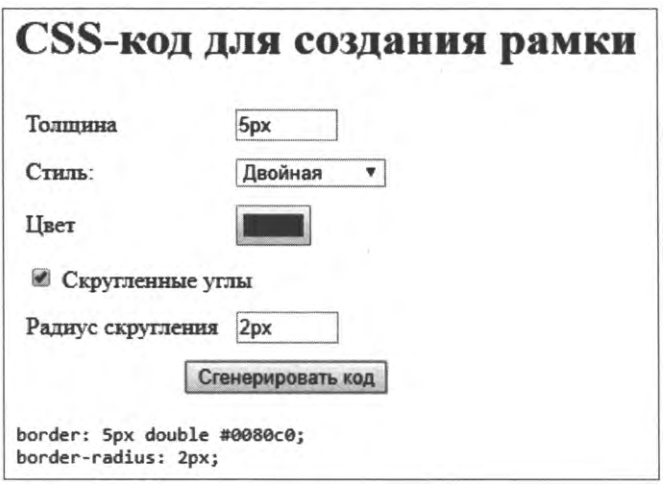


Рис. 9.2. CSS-код для задания рамки сгенерирован

9.3. Упражнение. Работаем с веб-формой

Код написанного в *разд. 9.2* приложения можно сделать компактнее, если заключить все элементы управления в веб-форму и получать введенные значения через нее. Давайте так и сделаем.

1. В папке 9\!sources сопровождающего книгу электронного архива (см. *приложение 3*) найдем страницу 9.1.html, скопируем ее в какое-либо место на локальном диске, переименуем в 9.2.html и откроем в текстовом редакторе.
2. Найдем тег <table>, формирующий таблицу, в которой находятся все элементы управления, и заключим его в веб-форму (в тег <form>):

```
<form>
    <table>
        <tr><td>Толщина</td>
        <td><input type="text" id="width" size="6" maxlength="6"
        value="thin"></td></tr>
        . . .
   </table>
</form>
```

3. Исправим HTML-код, создающий кнопку **Сгенерировать код** (исправленный код выделен полужирным шрифтом, удаленный — зачеркиванием):

```
<input type="buttonsubmit" id="generate" value="Сгенерировать код">
```

Мы изменили разновидность этой кнопки с обычной (`button`) на кнопку отправки данных (`submit`) и убрали якорь, который нам теперь не понадобится.

4. Добавим после закрывающего тега `<html>` сценарий, который получит доступ к форме и элементу `output`:

```
<script type="text/javascript">
    const form = document.forms[0];
    const output = document.getElementById('output');
</script>
```

Чтобы получить доступ к форме, мы воспользовались коллекцией `forms` объекта-страницы (см. *разд. 7.1*) — так проще.

5. Напишем код, делающий поле ввода **Радиус скругления** доступным или недоступным в зависимости от состояния флажка **Скругленные углы**:

```
form.elements[3].addEventListener('change', () => {
    form.elements[4].disabled = !form.elements[3].checked;
});
```

Свойство `elements` формы хранит коллекцию имеющихся в ней элементов управления. Мы использовали это свойство, чтобы получить доступ к флажку и полю ввода.

6. Код, генерирующий CSS-стиль, мы можем оформить как обработчик события `submit` формы. Это событие возникает перед отправкой данных, которая производится нажатием кнопки отправки данных (вот поэтому мы и изменили разновидность кнопки **Сгенерировать код** на *шаге 3*).

Напишем этот код:

```
form.addEventListener('submit', (evt) => {
    let s = 'border: ' + form.elements[0].value + ' ' +
        form.elements[1].options
        [form.elements[1].selectedIndex].value + ' ' +
        form.elements[2].value + ';\r\n';
    if (form.elements[3].checked)
        s += 'border-radius: ' + form.elements[4].value + ';';
    output.textContent = s;
    evt.preventDefault();
});
```

Доступ к элементам управления выполняем также через коллекцию `elements` формы. И не забываем отменить обработчик события `submit` по умолчанию — иначе форма выполнит отправку данных, что приведет к перезагрузке страницы в ее изначальном виде.

Откроем страницу 9.2.html в веб-обозревателе и проверим ее в работе.

9.3.1. Свойства, методы и события веб-формы

Веб-форма представляется классом `HTMLFormElement`. Помимо свойств, соответствующих атрибутам тегов, и свойств, методов и событий, унаследованных от суперкласса `HTMLElement`, он поддерживает еще несколько свойств (табл. 9.5), методов (табл. 9.6) и событий (табл. 9.7).

***Таблица 9.5.** Дополнительные свойства класса `HTMLFormElement`*

Свойство	Описание
`elements`	Коллекция всех элементов управления, имеющихся в форме
`length`	Количество элементов управления в форме
`autocomplete`	Если `'on'` — у всех полей ввода в форме выводится список ранее введенных значений для быстрого выбора (поведение по умолчанию). Если `'off'` — такой список не выводится

***Таблица 9.6.** Дополнительные методы класса `HTMLFormElement`*

Метод	Описание
`submit()`	Выполняет отправку данных, как если была нажата кнопка отправки данных
`reset()`	Выполняет сброс формы, как если бы была нажата кнопка сброса

***Таблица 9.7.** Дополнительные события класса `HTMLFormElement`*

Событие	Когда возникает?
Всплывающие	
`submit`	Перед отправкой данных (обработчик по умолчанию производит отправку данных)
`reset`	Перед сбросом формы (обработчик по умолчанию сбрасывает форму)

9.4. Упражнение. Реализуем валидацию данных в веб-форме

В приложении из *разд. 9.3* не помешает обезопаситься на тот случай, если пользователь забудет ввести толщину рамки, радиус скругления или же не укажет в них единицу измерения. Давайте сделаем это и пока ради простоты будем принимать лишь величины, заданные в пикселах.

1. В теги `<input>` страницы 9.2.html, создающие поля ввода **Толщина** и **Радиус скругления**, добавим атрибут без значения `required`, чтобы сделать эти поля

обязательными к заполнению (добавленный код выделен полужирным шрифтом):

```
<input type="text" id="width" . . . required>
. . .
<input type="text" id="radius" . . . required>
```

Это самая простая разновидность валидации, реализуемая средствами HTML.

Откроем исправленную страницу 9.2.html в веб-обозревателе, удалим значение из поля ввода **Толщина** и нажмем кнопку **Сгенерировать код**. Под полем ввода появится сообщение, требующее его заполнить (рис. 9.3).

Рис. 9.3. Сообщение, требующее заполнить поле

Такое же сообщение мы получим, если установим флажок **Скругленные углы**, чтобы сделать доступным поле ввода **Радиус скругления**, и нажмем кнопку **Сгенерировать код**, — HTML-валидация работает.

2. Приступим к созданию валидации, требующей указания единицы измерения и реализуемой программно средствами JavaScript.

> *Программную валидацию* следует производить в обработчике события `input` или `change` элемента управления.

Нам нужно проверить, заканчивается ли введенное значение толщины или радиуса скругления символами `px` (обозначение пикселов в стандарте CSS). Если это не так, следует пометить соответствующее поле ввода, как содержащее некорректное значение, задав для него текст сообщения об ошибке ввода.

> Для указания сообщения об ошибочном вводе достаточно вызвать у поля ввода метод `setCustomValidity` в следующем формате:
>
> `setCustomValidity(`*`<строка с сообщением об ошибке>`*`)`.

Если методу при вызове передать непустую строку, элемент управления считается содержащим некорректное значение. Чтобы пометить элемент управления как содержащий корректное значение, следует передать методу `setCustomValidity` пустую строку.

Итак, напишем функцию `checkValue`, которая выполнит необходимую проверку, и привяжем ее к полям ввода **Толщина** и **Радиус скругления** в качестве обработчика события `change`:

```
function checkValue() {
    if (this.value)
```

```
        if (this.value.endsWith('px'))
            this.setCustomValidity('');
        else
            this.setCustomValidity↵
            ('Укажите единицу измерения px (пикселы)');
}
form.elements[0].addEventListener('change', checkValue);
form.elements[4].addEventListener('change', checkValue);
```

В теле функции `checkValue` предварительно проверяем, занесено ли вообще значение в поле ввода. Если этого не сделать, метод `endsWith` не найдет в пустой строке символов `px`, и пользователь увидит сообщение, требующее указать единицу измерения, вместо стандартного **Заполните это поле**, что обескуражит его.

Обновим исправленную страницу, занесем в поле ввода **Толщина** значение без единицы измерения и нажмем кнопку **Сгенерировать код**. Под полем ввода появится сообщение, предлагающее дописать единицу измерения (рис. 9.4).

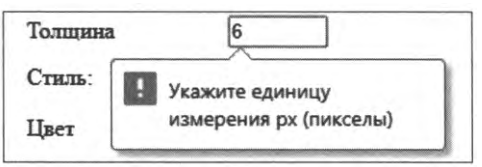


Рис. 9.4. Сообщение, предлагающее дописать единицу измерения

Для проверки допишем обозначение пикселов, вновь нажмем кнопку **Сгенерировать код** и посмотрим на появившийся ниже CSS-код стиля.

9.4.1. Вывод сообщений об ошибках в произвольном месте веб-страницы

По умолчанию веб-обозреватель выводит сообщения об ошибках ввода во всплывающих окошках под элементами управления. Но мы можем вывести их иным образом — например, в каком-либо элементе страницы.

Класс `HTMLInputElement`, представляющий поле ввода, поддерживает метод `checkValidity`. Он не принимает параметров и возвращает `true`, если в поле введено корректное значение, и `false` в противном случае.

Тот же класс поддерживает свойство `validationMessage`, хранящее строку с текстом сообщения об ошибке и доступное только для чтения.

Вот пример вывода сообщения об ошибке в элементе страницы `error`:

```
<input type="text" id="width" . . . required>
<div id="widthError"></div>
. . .
const width = document.getElementById('width');
const widthError = document.getElementById('widthError');
```

```
width.addEventListener('change', () => {
    if (width.checkValidity())
        widthError.textContent = 'OK!';
    else
        widthError.textContent = width.validationMessage;
});
```

Если требуется выполнить валидацию средствами JavaScript, выполняющий ее код следует поместить перед вызовом метода `checkValidity`:

```
width.addEventListener('change', () => {
    if (width.value)
        if (width.value.endsWith('px'))
            width.setCustomValidity('');
        else
            width.setCustomValidity↵
            ('Укажите единицу измерения px (пикселы)');
    if (width.checkValidity())
        widthError.textContent = 'OK!';
    else
        widthError.textContent = width.validationMessage;
});
```

Класс `HTMLInputElement` также поддерживает доступное только для чтения свойство `willValidate`. Оно хранит `true`, если в поле ввода будет выполняться валидация, и `false` — если не будет (это случается, если поле недоступно для ввода).

9.4.2. Вывод произвольных сообщений об ошибках

Класс `HTMLInputElement` поддерживает свойство `validity`, хранящее объект класса `ValidityState`. Он содержит сведения о том, какого рода ошибка была допущена при вводе значения.

Класс `ValidityState` поддерживает следующие свойства, доступные только для чтения:

- `valid` — `true`, если введенное значение полностью корректно, `false` в противном случае;
- `valueMissing` — `true`, если обязательное к заполнению поле ввода не заполнено, `false` в противном случае;
- `typeMismatch` — `true`, если введено значение неподходящего типа (например, строка, не являющаяся интернет-адресом, — в поле ввода интернет-адреса), `false` в противном случае;
- `tooLong` — `true`, если длина введенного значения превышает заданную в атрибуте тега `maxlength`, `false` в противном случае;
- `rangeUnderflow` — `true`, если число, занесенное в поле ввода числа, меньше указанного в атрибуте тега `min`, `false` в противном случае;

- `rangeOverflow` — `true`, если число, занесенное в поле ввода числа, больше указанного в атрибуте тега `max`, `false` в противном случае;
- `stepMismatch` — `true`, если число, занесенное в поле ввода числа, не попадает в интервал между значениями, заданный атрибутом тега `step`, `false` в противном случае;
- `patternMismatch` — `true`, если введенное значение не совпадает с шаблоном, заданным в атрибуте тега `pattern` (о нем разговор пойдет на *уроке 10*), `false` в противном случае;
- `customError` — `true`, если метод `setCustomValidity` был вызван с непустой строкой, `false` в противном случае.

Вот пример вывода произвольного сообщения об ошибке ввода:

```
<input type="text" id="width" . . . required>
<div id="widthError"></div>
. . .
const width = document.getElementById('width');
const widthError = document.getElementById('widthError');
width.addEventListener('change', () => {
    if (width.validity.valid)
        widthError.textContent = 'OK!';
    else if (width.validity.valueMissing)
        widthError.textContent = 'Задайте толщину рамки';
    else
        widthError.textContent = width.validationMessage;
});
```

9.5. Самостоятельные упражнения

- Доработайте веб-приложение 9.2.html так, чтобы при незаполнении поля **Толщина** или **Радиус скругления** выводилось сообщение «Задайте толщину» и «Задайте радиус скругления» соответственно.

  Подсказка: выполните необходимую проверку в функции `checkValue` и используйте тот факт, что у упомянутых ранее полей ввода заданы разные якоря.

- Если в раскрывающемся списке **Стиль** того же веб-приложения выбран пункт **Двойная** или любой из следующих за ним пунктов, пусть оно проверяет, чтобы толщина рамки была не менее 5 пикселов. Если же условие не выполнено, должно выводиться сообщение «Для этого стиля рамки укажите толщину не менее 5 пикселов».

- В папке 9\!sources сопровождающего книгу электронного архива (см. *приложение 3*) найдите страницу 9.5.1.html. На ее основе напишите веб-приложение (рис. 9.5), генерирующее CSS-код для формирования просветов на основе введенных пользователем данных (внутренние просветы создаются атрибутом стиля `padding`, внешние — `margin`). Реализуйте в приложении валидацию — провер-

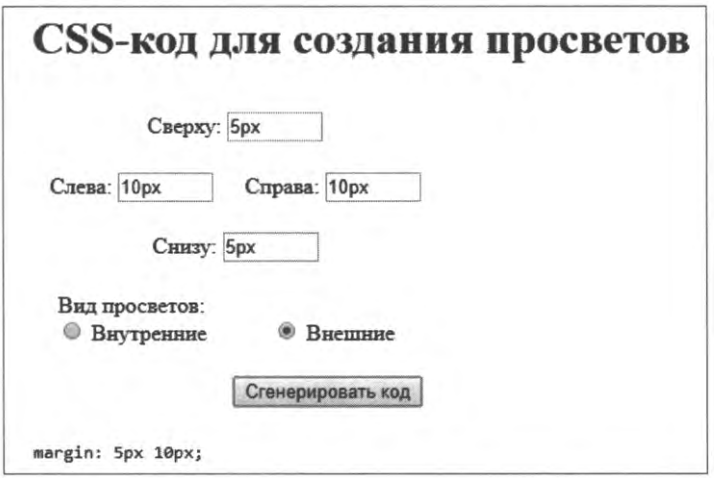


Рис. 9.5. Веб-приложение, генерирующее CSS-код для формирования просветов

ку на ввод всех четырех значений просветов и на указание в них единицы измерения `px`.

- В папке 9\!sources сопровождающего книгу электронного архива (см. *приложение 3*) найдите страницу 9.5.2.html. На ее основе напишите приложение-экзаменатор (рис. 9.6), предлагающее пользователю выбрать в списке все интерпретируемые языки программирования и нажать кнопку **Готово!**. В случае верного выбора внизу должна выводиться зеленая надпись «Правильно!», в случае неверного выбора — красная надпись «Неправильно!».

  Подсказка: для вывода надписи нужным цветом воспользуйтесь стилевыми классами `right` и `wrong`, уже имеющимися во внутренней таблице стилей этой страницы.

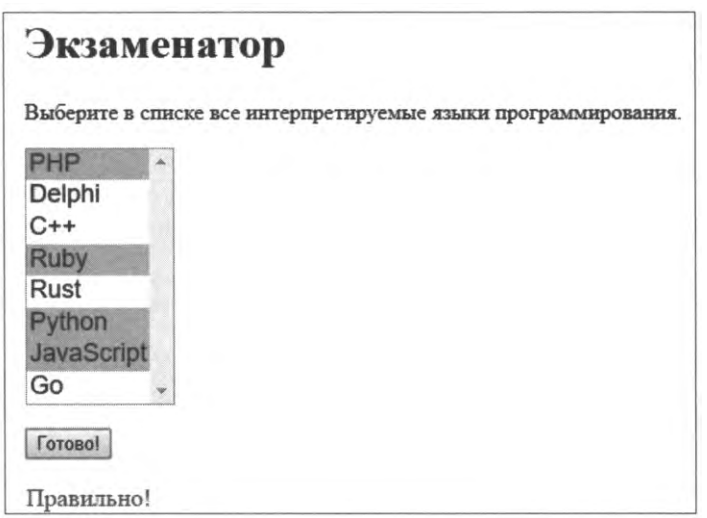


Рис. 9.6. Приложение-экзаменатор

Урок 10
Регулярные выражения

Регулярные выражения JavaScript
Литералы и модификаторы регулярных выражений
Поиск и обработка текста посредством регулярных выражений
Программная валидация с помощью регулярных выражений
HTML-валидация

10.1. Введение в регулярные выражения

Многие, если не большинство веб-сценариев занимаются обработкой строк. В том числе ищут в них фрагменты, совпадающие с определенными шаблонами, или проверяют совпадение с шаблонами отдельных строк (выполняют валидацию). Такие шаблоны оформляются в виде регулярных выражений.

> *Регулярное выражение* — шаблон, применяемый для поиска совпадающих с ним фрагментов текста и проверки на совпадение с ним заданных величин. Может включать в себя как обычные символы (которые обрабатываются как есть), так и литералы. Регулярное выражение представляет собой объект специализированного класса `RegExp`.

> *Литерал регулярного выражения* — символ или последовательность символов, имеющих особое значение. Литерал может обозначать символ определенной категории (например, любую букву, любую цифру, пробельный символ, букву из заданного набора и пр.), указывать количество повторений символа и др. Существует несколько разновидностей литералов.

10.1.1. Создание регулярных выражений

Создать регулярное выражение можно двумя способами:

- записав его между символами / (прямой слеш) в формате:

```
/<собственно шаблон>/[<модификаторы>].
```

> *Модификатор* — обозначение специфического режима обработки регулярного выражения, присутствие которого активирует этот режим. Имеет вид буквы латиницы.

Созданное регулярное выражение присваивается какой-либо переменной:

```
let rex = /CSS/;
```

Шаблон этого регулярного выражения содержит только обычные символы и совпадет с любой последовательностью символов `CSS`. Поиск совпадающих фрагментов ведется с учетом регистра символов:

```
// Здесь и далее фрагменты строк, совпадающие с регулярным выражением,
// выделены рамками
// Результат поиска: CSS CSS3 SCSS css HTML
```

Вот еще одно регулярное выражение

```
rex = /CSS\d/;
```

Литерал `\d`, присутствующий в нем, совпадает с любой цифрой от 0 до 9. Следовательно, весь шаблон совпадет с любой строкой вида: `CSS<`*цифра*`>`.

```
// Результат: CSS CSS3 SCSS css HTML
```

Модификатор `i` включает *поиск без учета регистра* символов:

```
rex = /CSS/i;
// Результат: CSS CSS3 SCSS css HTML
```

- создав объект класса `RegExp` посредством оператора `new`:

```
new RegExp(<шаблон>[, <модификаторы>]).
```

Шаблон указывается в виде обычной строки без символов `/`. *Модификаторы* также записываются в виде строки:

```
rex = new RegExp('CSS\d', 'i');
```

Этот способ имеет смысл применять лишь тогда, когда *шаблон* и (или) *модификаторы* создаются программно.

Для тестирования шаблонов регулярных выражений можно использовать интернет-службу «Regular expression» (рис. 10.1), доступную по адресу **https://regex101.com/**.

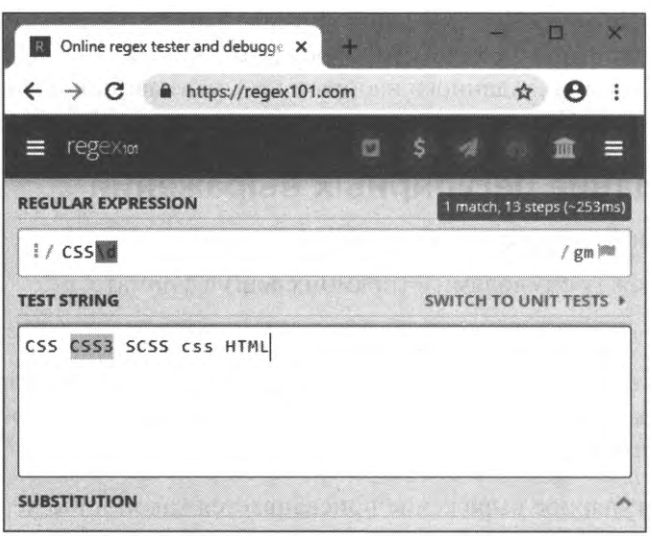


Рис. 10.1. Интернет-служба «Regular expression»

В поле ввода **Regular expression** заносится шаблон регулярного выражения, а в область редактирования **Test string** — обрабатываемая строка. Совпадающие фрагменты, найденные в обрабатываемой строке, подсвечиваются. Щелкнув на изображении флажка в правой части поля ввода **Regular expression**, можно вызвать меню для указания модификаторов.

10.1.2. Использование регулярных выражений: простые случаи

Проверить совпадение какой-либо строки с регулярным выражением или выполнить в строке поиск совпадающего фрагмента можно посредством двух методов:

- метод `test(<строка>)` класса `RegExp` возвращает `true`, если указанная *строка* совпадает с текущим регулярным выражением, и `false` в противном случае:

```
let rex = /CSS/;
let result = rex.test('CSS3');              // true
result = rex.test('HTML5');                 // false
```

- метод `search(<регулярное выражение>)` класса `String` ищет в текущей строке первый фрагмент, совпадающий с заданным *регулярным выражением*. Если совпадающий фрагмент найден, возвращается номер его первого символа (нумерация символов начинается с нуля), в противном случае: `-1`:

```
result = 'Язык CSS'.search(rex);            // 5
result = 'Язык HTML'.search(rex);           // -1
```

Метод `search` позволяет указывать в качестве параметра и обычную строку, которая автоматически преобразуется в регулярное выражение.

10.2. Литералы регулярных выражений

10.2.1. Метасимволы

> *Метасимвол* — литерал, обозначающий любой символ из какой-либо группы или символ с указанным кодом.

В табл. 10.1 приведен список поддерживаемых метасимволов.

***Таблица 10.1.** Список поддерживаемых метасимволов*

Литерал	Описание	Пример	Результат
. (точка)	Любой символ, кроме возврата каретки (`\r`) и перевода строки (`\n`)	`/./`	Basic!
`\w`	Алфавитно-цифровой символ (буква, цифра или подчеркивание)	`/\w/`	Basic!
`\W`	Не алфавитно-цифровой символ	`/\W/`	Basic!

Таблица 10.1 (окончание)

Литерал	Описание	Пример	Результат
\d	Цифра	/\d/	5px 20%
\D	Не цифра	/\D/	5px 20%
\s	Пробельный символ (пробел, табуляция, возврат каретки или перевод строки)	/\s/	5px 20%
\S	Не пробельный символ	/\S/	5px 20%
\b	Начало или конец слова	/\b3/	3px 3% W3C
\B	Не начало и не конец слова	/\B3/	3px 3% W3C
\u<*код*>	Символ с указанным Unicode-кодом	/\u043c/	Код символа

Примеры:

```
// Ищем отдельное слово вида 'CSS<цифра>'
rex = /\bCSS\d\b/;
result = rex.test('Язык CSS3');              // true
result = rex.test('Язык CSS');               // false
result = 'Пакет SCSS'.search(rex);           // -1

// Ищем обозначение цвета в формате RGB без учета регистра символов
rex = /#\w\w\w\w\w\w/i;
result = rex.test('color: #cb458a;');        // true
```

10.2.2. Поднаборы

Поднабор — литерал, обозначающий любой символ из заданного набора.

В табл. 10.2 приведен список поддерживаемых поднаборов.

***Таблица 10.2.** Список поддерживаемых поднаборов*

Литерал	Описание	Пример	Результат
[<*символы*>]	Любой из *символов*, приведенных в скобках	/[aeiuy]/	Basic
[^<*символы*>]	Любой из *символов*, *не* приведенных в скобках	/[^aeiuy]/	BaSiC
[<*c1*>-<*c2*>]	Любой из символов из диапазона от *c1* до *c2*	/[a-i]/	Field
[^<*c1*>-<*c2*>]	Любой из символов *не* из диапазона от *c1* до *c2*	/[^a-i]/	Field

Примеры:

```
rex = /[RST][i-m][^a-w]/;
result = rex.test('Sky');                    // true
result = rex.test('Tin');                    // false
```

10.2.3. Вариант

> *Вариант* — литерал, обозначающий любой фрагмент из перечисленных. Он записывается в формате:
>
> *<фрагмент 1>|<фрагмент 2>| . . . |<фрагмент n-1>|<фрагмент n>.*

В состав фрагментов включаются все символы и литералы, расположенные между вертикальными линиями, началом и концом регулярного выражения.

Примеры:

```
// Ищем строки 'jpg', 'jpeg' или 'jpe' без учета регистра
rex = /jpg|jpeg|jpe/i;
result = 'logo.jpg'.search(rex);            // 5
result = 'background.jpeg'.search(rex);     // 11
// Ищем либо 'HTML<цифра>', либо 'JavaScript'
rex = /HTML\d|JavaScript/;
result = 'Язык HTML5'.search(rex);          // 5
result = 'Учите JavaScript!'.search(rex);   // 6
result = 'Язык HTML'.search(rex);           // -1
```

10.2.4. Квантификаторы

> *Квантификатор* — литерал, указывающий количество повторений или местоположение символа.

Вот все поддерживаемые квантификаторы:

- *<символ>*+ — *символ* может присутствовать в произвольном количестве экземпляров:

```
rex = /\bCS+S\b/i;  // Результат: CS CSS CSSS CSSSSSSSS
```

- *<символ>** — *символ* может присутствовать в произвольном количестве экземпляров или вообще отсутствовать:

```
rex = /\bCS*S\b/i;  // Результат: CS CSS CSSS CSSSSSSSS
```

- *<символ>*? — *символ* может присутствовать в одном экземпляре или отсутствовать:

```
rex = /\bCS?S\b/i;  // Результат: CS CSS CSSS CSSSSSSSS
```

- *<символ>*{*<n>*} — *символ* должен присутствовать в количестве *n* экземпляров:

```
rex = /\bCS{2}S\b/i;
// Результат: CS CSS CSSS CSSSSSSSS
```

- *<символ>*{*<n>*,} — *символ* должен присутствовать в количестве не менее *n* экземпляров:

```
rex = /\bCS{2,}S\b/i;
// Результат: CS CSS CSSS CSSSSSSSS
```

- *<символ>{<n1>,<n2>}* — *символ* должен присутствовать в количестве от *n1* до *n2* экземпляров:

```
rex = /\bCS{1,2}S\b/i;
// Результат: CS CSS CSSS CSSSSSSSS
```

- *^<символ>* — *символ* должен находиться в начале текста:

```
rex = /^[0-9]+../;      // Результат: 20pt 30px 4% 200mm
```

Если в регулярном выражении указан модификатор `m` (будет рассмотрен далее), этот квантификатор также обозначает начало отдельной строки текста;

- *<символ>$* — *символ* должен находиться в конце текста:

```
rex = /[0-9]+..$/;      // Результат: 20pt 30px 4% 200mm
```

Если в регулярном выражении указан модификатор `m` (будет рассмотрен далее), этот квантификатор также обозначает конец отдельной строки текста;

- *<символ 1>(?=<символ 2>)* — за *символом 1* должен следовать *символ 2*, причем последний не включается в состав фрагмента, совпадающего с регулярным выражением:

```
// Ищем последовательность цифр, запятых и точек, за которой следует
// произвольное количество пробелов и строка 'руб',
// т. е. цену в рублях.
// Чтобы поместить в регулярное выражение символ точки, мы предварим
// его обратным слешем. Если этого не сделать, веб-обозреватель
// посчитает точку за литерал ., совпадающий с любым символом.
rex = /[\d,\.]+(?=\s*руб)/;
// Результат: 12 12 руб 34,67 руб 89.8 руб 89.8
```

- *<символ 1>(?!<символ 2>)* — за *символом 1 не* должен следовать *символ 2*, причем последний не включается в состав фрагмента, совпадающего с регулярным выражением:

```
// Ищем произвольное количество цифр, за которыми не следует
// комбинация из произвольного количества цифр и символов 'px',
// т. е. величины, указанные не в пикселах
rex = /\d+(?!\d*px)/;  // Результат: 20pt 30px 4% 200mm
```

10.2.5. Подквантификатор

По умолчанию квантификаторы `+` и `*` работают так, чтобы совпавшей с регулярным выражением оказалась строка максимальной длины (*«жадный» режим поиска*). В качестве примера рассмотрим регулярное выражение, которое совпадает с последовательностью любых символов, взятых в апострофы:

```
rex = /'.*'/;
// Результат: c++ 'html' 'css' 'javascript' python
```

Видно, что регулярное выражение, работая в «жадном» режиме, «захватило» самый длинный фрагмент — расположенный между первым и последним апострофами. Чтобы оно «захватывало», напротив, наиболее короткий фрагмент, его следует переключить в *«щедрый» режим поиска*.

> *Подквантификатор* — литерал, указываемый после квантификатора и изменяющий режим его работы.

Подквантификатор `?`, указанный после квантификатора `+` и `*`, переключает его в «щедрый» режим:

```
rex = /'.*?'/;
// Результат: c++ 'html' 'css' 'javascript' python
```

Будучи указанным в другом месте регулярного выражения, символ вопросительного знака считается квантификатором `?`, описанным ранее.

10.2.6. Группы и обратные ссылки

> *Группа* — часть регулярного выражения, взятая в круглые скобки. Группа может быть любой длины, но всегда ведет себя как один символ.

Примеры:

```
// Регулярное выражение, совпадающее со строками 'jpg', 'jpeg' и 'jpe'.
// Группа не использовалась.
rex = /jpg|jpeg|jpe/;
// То же самое регулярное выражение, но записанное с применением группы.
// Получилось короче.
rex = /jp(g|eg|e)/;
// Регулярное выражение, совпадающее с последовательностью цифр,
// после которой может присутствовать строка 'руб'
rex = /\d+(руб)?/;
```

Фрагмент, совпавший с группой, запоминается веб-обозревателем:

```
// Фрагмент, совпавший с группой и запомненный веб-обозревателем,
// здесь и далее выделен черной заливкой
rex = /(\d+)px/;                    // Результат: 20px 40pt
```

> *Обратная ссылка* — литерал регулярного выражения, ссылающийся на фрагмент, который совпадает с определенной группой.
>
> Обратная ссылка записывается в формате:
>
> *\<порядковый номер группы>*.
>
> Группы в регулярном выражении нумеруются в порядке слева направо, начиная с 1.

Далее приведен пример регулярного выражения, совпадающего с любым парным тегом. Внутри него созданы две группы: первая — для имени тега, вторая — для его содержимого. В записи закрывающего тега применяется обратная ссылка, хра-

нящая имя тега из первой группы. Чтобы поместить в регулярное выражение символ прямого слеша, мы предварили его обратным слешем:

```
rex = /<(\w+)>(.*?)<\/\1>/;          // Результат: <strong>HTML</strong>
```

10.2.7. Обычные символы

Все прочие символы в шаблонах регулярных выражений обозначают сами себя.

Если требуется вставить в шаблон символ, совпадающий с каким-либо литералом, его следует предварить знаком обратного слеша (\):

```
// Точка в шаблоне регулярного выражения — это литерал,
// обозначающий любой символ
const rex2 = /bhv.ru/;
// Результат: bhv:ru bhv.ru bhv ru
// Чтобы вставить в шаблон символ точки, его нужно предварить обратным слешем
const rex3 = /bhv\.ru/;
// Результат: bhv:ru bhv.ru bhv ru
// Чтобы вставить в регулярное выражение прямой слеш, обратный слеш
// или звездочку, обозначающих различные литералы,
// их также предваряют обратным слешем: \/, \\, \*
```

10.3. Поиск и обработка фрагментов, совпадающих с регулярными выражениями

10.3.1. Обычный режим поиска

> В *обычном режиме*, используемом по умолчанию, поиск останавливается после нахождения первого фрагмента, совпадающего с заданным регулярным выражением.

Метод `exec(<строка>)` класса `RegExp` ищет в *строке* фрагмент, совпадающий с текущим регулярным выражением, и возвращает результат поиска в виде массива. Первый (с индексом `0`) элемент этого массива хранит найденный фрагмент. Помимо этого, в объекте массива есть добавленные свойства:

- `index` — номер первого символа совпавшего фрагмента в исходной *строке*;
- `input` — сама исходная *строка*.

Если поиск завершился неудачей, метод возвращает `null`.

Пример:

```
rex = /\d+px/;
result = rex.exec('5px 4pt 10px');
let s1 = result[0];                  // '5px'
let ind = result.index;              // 0
let s = result.input;                // '5px 4pt 10px'
```

Класс RegExp поддерживает свойство lastIndex, хранящее номер символа строки, с которого начнется поиск, и доступное для чтения и записи.

Вот пример использования этого свойства:

```
rex.lastIndex = 5;
result = rex.exec('5px 4pt 10px');
s1 = result[0];                          // '10px'
ind = result.index;                      // 8
```

Если в шаблоне регулярного выражения присутствуют группы, фрагменты, совпадающие с ними, будут храниться в последующих (с индексами больше 0) элементах массива, возвращенного методом:

```
rex = /<(\w+)>(.*?)<\/\1>/;
result = rex.exec('HTML <strong>CSS</strong> JavaScript');
s1 = result[0];                          // '<strong>CSS</strong>'
let s2 = result[1];                      // 'strong'
let s3 = result[2];                      // 'CSS'
```

Метод match(*<регулярное выражение>*) класса String ищет в текущей строке фрагмент, совпадающий с *регулярным выражением*, и возвращает результат поиска в виде массива, аналогичного выдаваемому методом exec класса RegExp (см. ранее):

```
rex = /\d+px/;
result = '5px 4pt 10px'.match(rex);
s1 = result[0];                          // '5px'
ind = result.index;                      // 0
s = result.input;                        // '5px 4pt 10px'
```

Если поиск завершился неудачей, метод также возвращает null.

10.3.2. Глобальный поиск

> При *глобальном* поиске методы exec класса RegExp и match класса String ищут в строке все совпадающие фрагменты.

Глобальный поиск устанавливается указанием в регулярном выражении модификатора g.

В случае глобального поиска:

- метод exec класса RegExp — находит первый совпавший фрагмент, завершает работу с возвратом результата и заносит в свойство lastIndex регулярного выражения номер символа, располагающегося сразу за найденным фрагментом. При следующем вызове этого метода будет выполнен поиск, начиная с этого места, в результате чего будет найден следующий фрагмент, и т. д.:

```
rex = /\d+px/g;
result = rex.exec('5px 4pt 10px');
s1 = result[0];                          // '5px'
ind = result.index;                      // 0
let li = rex.lastIndex;                  // 3
```

```
result = rex.exec('5px 4pt 10px');
s1 = result[0];                              // '10px'
ind = result.index;                          // 8
li = rex.lastIndex;                          // 12
result = rex.exec('5px 4pt 10px');
// Здесь метод exec вернет значение null — поиск закончен
```

- метод `match` класса `String` — возвращает обычный массив со всеми найденными совпадениями (фрагменты, совпадающие с группами, в него не включаются):

```
result = '5px 4pt 10px'.match(rex);
s1 = result[0];                              // '5px'
s2 = result[1];                              // '10px'
```

10.3.3. Многострочный поиск

> При *многострочном* поиске квантификаторы `^` и `$` обозначают не только начало и конец текста, но и начало и конец каждой отдельной строки в тексте.

Отдельные строки в строковом значении разделяются комбинацией символов `\r\n` (реже — отдельными символами `\r` или `\n`):

```
let s = 'Первая строка\r\nВторая строка\r\nТретья строка';
```

Пример:

```
rex = /^[0-9]+../;       // Результат: 20pt 30px\r\n200mm 5cm
rex = /^[0-9]+../m;      // Результат: 20pt 30px\r\n200mm 5cm
rex = /[0-9]+..$/;       // Результат: 20pt 30px\r\n200mm 5cm
rex = /[0-9]+..$/m;      // Результат: 20pt 30px\r\n200mm 5cm
```

10.3.4. Замена совпавших фрагментов

Метод `replace` класса `String`, знакомый нам по *разд. 4.4*, может выполнять замену фрагментов, совпавших с указанным *регулярным выражением*:

```
replace(<регулярное выражение>, <заменяющая подстрока>)
```

Пример:

```
rex = /CSS\d/;
result = 'HTML CSS5 JavaScript'.replace(rex, 'CSS');
                                             // 'HTML CSS JavaScript'
```

Применив регулярное выражение с модификатором `g`, можно заменить все совпавшие фрагменты, а не только первый встреченный:

```
result = '5px 4pt 10px'.replace(/px/g, 'mm');
                                             // '5mm 4pt 10mm'
```

В заменяющей подстроке можно использовать следующие специализированные литералы, работающие только здесь:

- `$&` — обозначает весь совпавший фрагмент;
- `$<gn>` — фрагмент из группы с номером *gn* (обратная ссылка);
- `` $` `` — часть строки до совпавшего фрагмента;
- `$'` — часть строки после совпавшего фрагмента;
- `$$` — символ доллара (`$`).

Примеры:

```
result = 'HTML, CSS, JavaScript'.↵
replace(/\b\w+\b/g, '<em>$&</em>');
                    // '<em>HTML</em>, <em>CSS</em>, <em>JavaScript</em>'
result = '5px 4pt 20%'.replace(/(\d+)(px|pt)/g, '$1mm');
                                              // '5mm 4mm 20%'
```

10.3.5. Прочие полезные инструменты

Метод `split` класса `String`, описанный в *разд. 4.4*, может разбить строку, руководствуясь указанным в качестве символа-разделителя *регулярным выражением*:

```
split(<регулярное выражение>[, <предельный размер массива>])
```

Пример:

```
result = 'HTML,CSS;JavaScript'.split(/,|;/);
                                        // ['HTML', 'CSS', 'JavaScript']
```

Класс `RegExp` поддерживает еще несколько свойств (помимо `lastIndex`), которые могут оказаться полезными и которые доступны только для чтения:

- `source` — шаблон регулярного выражения без символов слеша и модификаторов, представленный в виде строки;
- `global` — `true`, если установлен модификатор `g`, `false` в противном случае;
- `ignoreCase` — `true`, если установлен модификатор `i`, `false` в противном случае;
- `multiline` — `true`, если установлен модификатор `m`, `false` в противном случае.

10.4. Упражнение. Выполняем программную валидацию с помощью регулярного выражения

Веб-приложение, написанное в *разд. 9.4*, позволяет задать в качестве единицы измерения только пикселы. Доработаем его, чтобы можно было указывать также пункты и проценты. И используем для этого регулярное выражение.

1. Найдем в папке 10\!sources сопровождающего книгу электронного архива (см. *приложение 3*) страницу 9.2.html, скопируем ее куда-либо на локальный диск и откроем копию в текстовом редакторе.

2. В начало кода сценария добавим строку, создающую регулярное выражение и присваивающее ее переменной `rex`:

```
const rex = /^[0-9]+(px|pt|%)$/i;
```

Это регулярное выражение будет совпадать с любыми значениями формата:

*<последовательность цифр от 0 до 9>*px|pt|%

Модификатор `i` включает поиск без учета регистра символов.

3. Исправим код функции `checkValue` следующим образом (исправления выделены полужирным шрифтом):

```
function checkValue() {
    if (this.value)
        if (rex.test(this.value))
            this.setCustomValidity('');
        else
            this.setCustomValidity('Введите корректное значение');
}
```

Откроем страницу 9.2.html в веб-обозревателе и проверим, как она работает.

10.5. HTML-валидация с применением регулярных выражений

Чтобы выполнить валидацию введенного в поле значения на совпадение с регулярным выражением, необязательно писать веб-сценарий. Достаточно указать нужное регулярное выражение в атрибуте `pattern` тега `<input>`. Регулярное выражение должно быть записано без символов слеша и без модификаторов. Пример:

```
<input type="text" id="width" pattern="[0-9]+(px|pt|%)">
```

Атрибут тега `pattern` поддерживается в обычном поле ввода, полях ввода пароля, интернет-адреса и адреса электронной почты.

Недостатки этого способа валидации: необходимость записи регулярного выражения в каждое поле ввода, в котором нужно выполнить проверку значения, и невозможность указания модификаторов (вследствие чего, например, нельзя выполнить сравнение без учета регистра символов).

10.6. Самостоятельные упражнения

- В сценарий страницы 9.2.html, выполняющий валидацию (см. *разд. 10.4*), добавьте поддержку остальных единиц измерения CSS: миллиметров (`mm`), сантиметров (`cm`), дюймов (`in`), пик (`pc`), `em`, `vw` и `vh`, а также предопределенных значений толщины: `thin`, `medium` и `thick`. Заодно переименуйте функцию `checkValue` в `checkWidth`.

- Там же объявите функцию `checkRadius`, проверяющую на корректность величину радиуса скругления. Она будет полностью аналогична функции `checkWidth`, за исключением поддержки предопределенных значений: `thin`, `medium` и `thick`.
- Вынесите функции `checkWidth` и `checkRadius` в файл веб-сценария checker.js, создав тем самым библиотеку, пригодную для использования при разработке других решений. Добавьте в эту библиотеку функции `prepareCheckWidth` и `prepareCheckRadius`, которые выполнят привязку функций `checkWidth` и `checkRadius` к событию `change` элемента страницы, переданного в единственном параметре. Проверьте библиотеку, используя страницу 9.2.html.

Урок 11
Взаимодействие с веб-обозревателем

Работа с окном веб-обозревателя
Понятие BOM
Получение текущего интернет-адреса
Получение сведений о локальном компьютере
Вывод стандартных диалоговых окон

11.1. Работа с окном веб-обозревателя

Мы можем выяснить местоположение и размеры окна веб-обозревателя, величину, на которую была прокручена открытая в нем страница, принудительно прокрутить страницу, отреагировать на завершение загрузки страницы, ее прокрутку, изменение размеров окна и т. п.

Все это выполняется посредством свойств (табл. 11.1), методов (табл. 11.2) и событий (табл. 11.3) класса `Window`, который представляет окно веб-обозревателя. Объект этого класса, представляющий текущее окно, хранится в переменной `window`, которая создается самим веб-обозревателем и доступна в любом веб-сценарии.

Класс `Window`, равно как и все прочие классы BOM, что мы здесь рассмотрим, не является потомком класса `HTMLElement`.

> *BOM* (Browser Object Model, объектная модель веб-обозревателя) — набор объектов, представляющих окно веб-обозревателя, текущий интернет-адрес и сведения о клиентском компьютере.

Таблица 11.1. *Свойства класса* `Window`

Свойство	Описание
`innerWidth`	Ширина внутренней области окна
`innerHeight`	Высота внутренней области окна
`outerWidth`	Ширина всего окна
`outerHeight`	Высота всего окна
`screenLeft`	Горизонтальная координата левого верхнего угла окна относительно левого верхнего угла экрана

***Таблица 11.1** (окончание)*

Свойство	Описание
screenTop	Вертикальная координата левого верхнего угла окна относительно левого верхнего угла экрана
screenX	То же самое, что и screenLeft
screenY	То же самое, что и screenTop
pageXOffset	Величина, на которую открытая в окне страница была прокручена по горизонтали
pageYOffset	Величина, на которую открытая в окне страница была прокручена по вертикали
scrollX	То же самое, что и pageXOffset
scrollY	То же самое, что и pageYOffset

Свойства класса Window, приведенные в табл. 11.1, хранят числовые величины, измеряемые в пикселах, и доступны только для чтения.

***Таблица 11.2. Методы класса** Window*

Метод	Описание
blur()	Деактивирует текущее окно
focus()	Активирует текущее окно
print()	Открывает стандартное диалоговое окно печати

scrollBy(<*dx*>, <*dy*>)	Прокручивает открытую в текущем окне страницу на *dx* пикселов по горизонтали и *dy* пикселов по вертикали. Положительные значения вызывают прокрутку вправо и вниз, отрицательные — влево и вверх. Примеры:
	`// Прокручиваем страницу на 200 пикселов вправо`
	`window.scrollBy(200, 0);`
	`// Прокручиваем страницу на 300 пикселов вверх`
	`window.scrollBy(0, -300);`

scrollTo(<*x*>, <*y*>)	Прокручивает открытую в текущем окне страницу в позицию с указанными в пикселах координатами *x* и *y*:
	`// Прокручиваем страницу в позицию [0, 500]`
	`window.scrollTo(0, 500);`

***Таблица 11.3. События класса** Window*

Событие	Когда возникает?
load	После загрузки страницы
pageshow	После загрузки страницы и события load

Таблица 11.3 (окончание)

Событие	Когда возникает?
scroll	При прокрутке содержимого страницы
resize	При изменении размеров окна
focus	При активизации окна
blur	При деактивизации окна
beforeunload	Перед уходом с текущей страницы
pagehide	Перед уходом с текущей страницы, после события beforeunload
beforeprint	Перед открытием стандартного диалогового окна печати
afterprint	После закрытия стандартного диалогового окна печати страницы нажатием кнопки **ОК** или **Отмена**
hashchange	При переходе на другой якорь

Помимо этого, в окне можно обрабатывать любые всплывающие события, возникающие в элементах страницы. Вот пример обработки в окне события click:

```
window.addEventListener('click', () => {
    // Обрабатываем событие click
});
```

11.1.1. Еще один способ записи веб-сценариев, манипулирующих элементами страницы

Чтобы веб-сценарий, манипулирующий элементами страницы, смог работать, к моменту его исполнения в памяти уже должны присутствовать объекты, представляющие эти элементы. Поэтому такие сценарии часто помещают в конец HTML-кода страницы:

```
<html>
    . . .
</html>
<script type="text/javascript">
    // Здесь помещается код сценария
</script>
```

Однако подобный сценарий можно оформить и как обработчик события load окна. Это событие возникает, как только страница загружена и обработана, следовательно, его обработчик исполняется, когда в памяти уже сформированы все объекты, представляющие элементы страницы. Такой сценарий можно поместить куда угодно — например, в секцию заголовка страницы:

```
<html>
    <head>
        . . .
```

```
        <script type="text/javascript">
            window.addEventListener('load', () => {
                // Здесь помещается код сценария
            });
        </script>
    </head>
    <body>
        . . .
    </body>
</html>
```

Этот подход часто применяется, если к странице привязано несколько сценариев: основной и используемые в нем библиотеки. Тогда все необходимые теги `<script>` помещаются в одном месте HTML-кода (обычно в секции заголовка), что упрощает сопровождение:

```
<head>
    . . .
    <!-- Библиотеки -->
    <script type="text/javascript" src="scripts/library1.js"></script>
    <script type="text/javascript" src="scripts/library2.js"></script>
    . . .
    <!-- Основной сценарий -->
    <script type="text/javascript">
        window.addEventListener('load', () => {
            . . .
        });
    </script>
</head>
```

11.2. Упражнение. Навигация по якорям с подсветкой активной гиперссылки

В *разд. 7.4* мы создали страницу, включающую элемент с прокруткой и панель навигации. При щелчках на гиперссылках содержимое элемента прокручивалось таким образом, чтобы соответствующий раздел становился видимым. Давайте сделаем аналогичную страницу, в которой прокручиваться будет все ее содержимое, а для подсветки активной гиперссылки станем обрабатывать событие `hashchange` окна.

1. В папке 11\!sources сопровождающего книгу электронного архива (см. *приложение 3*) найдем страницу 11.2.html (рис. 11.1) и скопируем ее в какое-либо место на жестком диске. Эта страница аналогична той, что мы создали в *разд. 7.4*.
2. Откроем страницу 11.2.html в текстовом редакторе и допишем в конце ее HTML-кода сценарий с объявлением функции `hightlightActiveLink`, которая получит с единственным параметром якорь и подсветит гиперссылку, в которой он записан:

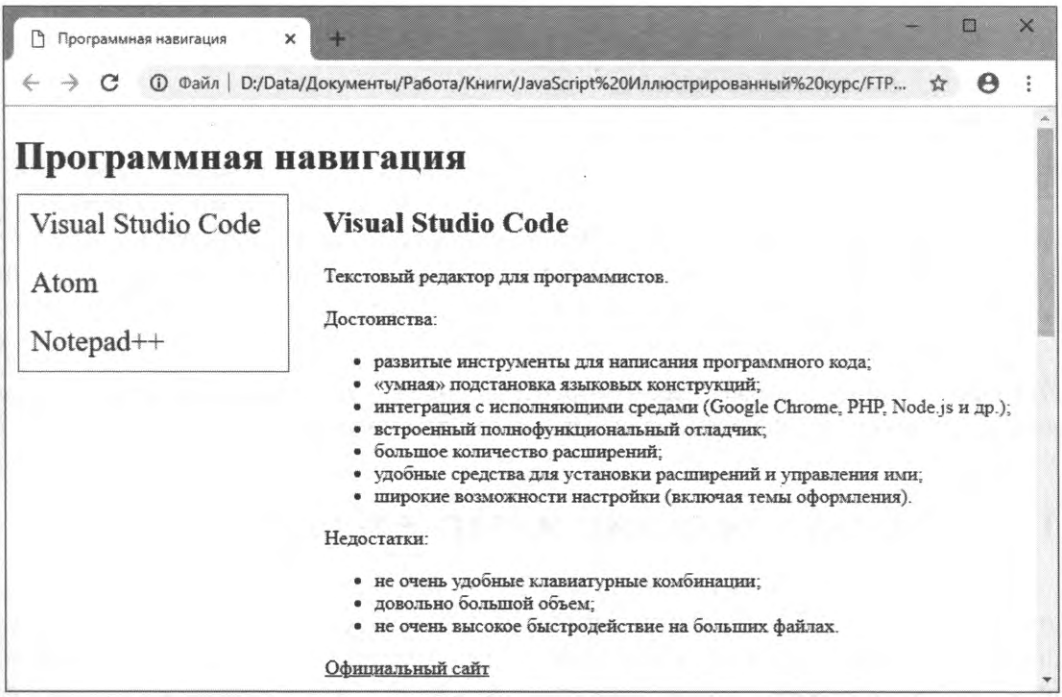


Рис. 11.1. Веб-страница, включающая элемент с прокруткой и панель навигации

```
<script type="text/javascript">
    function highlightActiveLink(hash) {
        active = document.querySelector('nav a.active');
        if (active)
            active.classList.remove('active');
        active = document.querySelector('nav a[href$=' + hash + ']');
        active.classList.add('active');
    }
</script>
```

Сначала ищем активную в текущий момент гиперссылку и, если она есть, убираем у нее стилевой класс `active`, чтобы снять подсветку. Потом ищем гиперссылку с полученным в параметре якорем и привязываем к ней стилевой класс `active`, тем самым подсвечивая.

3. Добавим обработчик события `hashchange` окна, который извлечет из интернет-адреса якорь, на который был выполнен переход, и передаст его функции `hightlightActiveLink`:

```
window.addEventListener('hashchange', (evt) => {
    const hashIndex = evt.newURL.indexOf('#');
    let hash;
    if (hashIndex >= -1)
        hash = evt.newURL.substr(hashIndex + 1);
```

```
    else
        hash = 'vsc';
    highlightActiveLink(hash);
});
```

В полученном из свойства `newURL` объекта события интернет-адресе ищем символ решетки (`#`), с которого начинается якорь. Если таковой есть, вырезаем из интернет-адреса имя якоря без символа решетки. Если же решетки в интернет-адресе нет, берем в качестве якоря активной гиперссылки `vsc` — это якорь самой первой гиперссылки в панели навигации. После чего вызываем функцию `hightlightActiveLink`, передав ей полученный якорь.

Откроем страницу 11.2.html в веб-обозревателе, пощелкаем по гиперссылкам и убедимся, что активная гиперссылка каждый раз подсвечивается.

11.3. Работа с текущим интернет-адресом

Есть возможность получить интернет-адрес текущей страницы, равно как и его составные части (адрес хоста, путь к файлу и пр.), программно перейти по другому интернет-адресу и перезагрузить страницу. Это выполняет класс `Location`, который представляет текущий интернет-адрес, и объект которого хранится в свойстве `location` объекта окна.

Свойства класса `Location` (все, за исключением одного) доступны для чтения и записи:

- `href` — полный интернет-адрес в виде строки:

```
let href = window.location.href;
                // 'http://www.dm.ru:8000/pages/page.html?par=12#anch'

// Выполняем программный переход по другому интернет-адресу
location.href = 'https://www.google.ru/';
```

- `origin` — строка вида *<обозначение протокола><адрес хоста>*[:*<номер TCP-порта>*], причем *номер TCP-порта* присутствует только в том случае, если он указан в интернет-адресе:

```
let org = location.origin;          // 'http://www.dm.ru:8000'
```

  Это свойство доступно только для чтения;

- `host` — сочетание вида *<адрес хоста>*[:*<номер TCP-порта>*], причем *номер TCP-порта* присутствует только в том случае, если он указан в интернет-адресе:

```
let host = location.host;           // 'www.dm.ru:8000'
```

- `protocol` — обозначение протокола с завершающим символом двоеточия, но без двойного слеша:

```
let protocol = location.protocol;   // 'http:'
```

- `hostname` — адрес хоста:

```
let hn = location.hostname;          // 'www.dm.ru'
```

- `port` — номер TCP-порта в виде строки. Если порт в интернет-адресе не указан, хранит пустую строку:

```
let port = location.port;            // '8000'
```

- `pathname` — путь к файлу с начальным символом слеша. Если путь не указан, хранит строку со слешем (`'/'`):

```
let path = location.pathname;        // '/pages/page.html'
```

- `search` — строка с GET-параметрами, включающая начальный вопросительный знак. Если таковые в интернет-адресе отсутствуют, хранит пустую строку:

```
let search = location.search;        // '?par=12'
```

- `hash` — имя якоря с начальным символом решетки. Если якоря нет, хранит пустую строку:

```
let hash = location.hash;            // '#anch'

// Если присвоить этому свойству другой якорь (уже без начального
// символа решетки!), будет выполнен переход по этому якорю
location.hash = 'anch2';
```

Также поддерживаются следующие три метода:

- `assign(<`*интернет-адрес*`>)` — выполняет переход на страницу, находящуюся по указанному *интернет-адресу*:

```
location.assign('https://www.google.ru/');
```

- `replace(<`*интернет-адрес*`>)` — то же самое, что и `assign`, только текущий интернет-адрес не сохраняется в истории веб-обозревателя;

- `reload([<`*с веб-сервера?*`>])` — перезагружает текущую страницу. Если передать с параметром значение `false` или вообще опустить параметр, страница, по возможности, будет загружена из кеша веб-обозревателя. Если же задать `true`, страница будет загружена с веб-сервера, даже если она имеется в кеше:

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

11.4. Получение сведений о клиентском компьютере

Узнать сведения об экранной подсистеме клиентского компьютера позволит объект класса `Screen`, хранящийся в свойстве `screen` объекта окна. Его свойства приведены в табл. 11.4 — все они хранят числовые величины и доступны только для чтения.

Таблица 11.4. *Свойства объекта класса* `Screen`

Свойство	Описание
`availWidth`	Ширина свободной области экрана (без учета панели задач) в пикселах
`availHeight`	Высота свободной области экрана (без учета панели задач) в пикселах
`width`	Ширина всего экрана в пикселах
`height`	Высота всего экрана в пикселах
`colorDepth`	Глубина цвета в битах на пиксел

Пример:

```
<img id="logo">
. . .
const logo = document.getElementById('logo');
if (screen.availWidth > 1920)
    logo.src = '/images/logo_hi_res.png';
else
    logo.src = '/images/logo.png';
```

11.5. Вывод стандартных диалоговых окон

Для вывода стандартных диалоговых окон сообщения, предупреждения и запроса на ввод значения применяются три следующих метода класса `Window`:

- `alert(<`*текст сообщения*`>)` — выводит окно-сообщение с указанным *текстом* и кнопкой **OK** (рис. 11.2):

```
window.alert('Введите неотрицательное число');
```

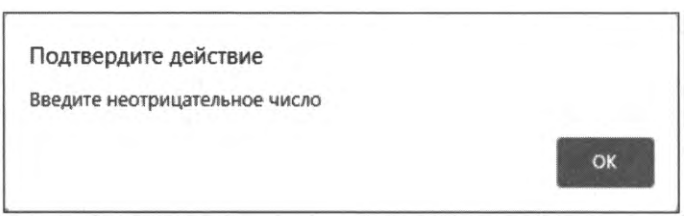


Рис. 11.2. Окно-сообщение с текстом и кнопкой **OK**

- `confirm(<`*текст сообщения*`>)` — выводит окно-предупреждение с указанным *текстом* и кнопками **OK** и **Отмена** (рис. 11.3). Возвращает `true`, если пользователь нажал кнопку **OK**, `false` — если нажал кнопку **Отмена**:

```
window.confirm('Выполнить операцию?');
```

- `prompt(<`*текст сообщения*`>[, <`*изначальное значение*`>])` — выводит окно с запросом на ввод значения, содержащее указанный *текст*, поле ввода, куда заносится

требуемое значение, и кнопки **ОК** и **Отмена** (рис. 11.4). Если было задано *изначальное значение*, оно будет подставлено в поле ввода. Возвращает введенное значение в виде строки, если пользователь нажал кнопку **ОК**, и `null` — если была нажата кнопка **Отмена**:

```
window.prompt('Введите свое имя', 'Вася Пупкин');
```

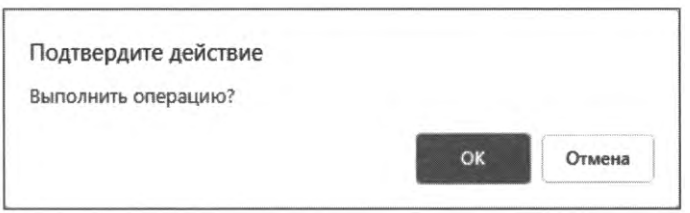


Рис. 11.3. Окно-предупреждение с текстом и кнопками **ОК** и **Отмена**

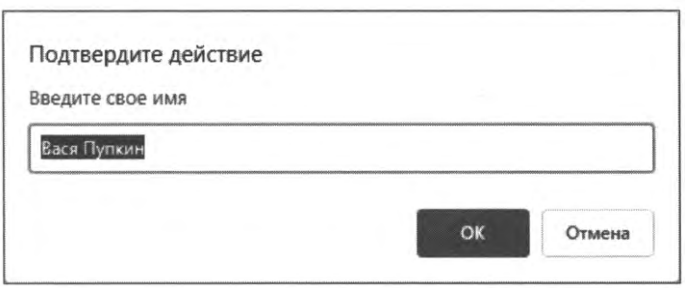


Рис. 11.4. Окно с запросом на ввод значения, содержащее текст, поле ввода и кнопки **ОК** и **Отмена**

Метод `prompt` в настоящее время применяется только при обучении программированию (мы сами применяли его для этой цели в предыдущих уроках), экспериментах и разработке прототипов. Использование его в рабочих веб-сайтах считается плохим стилем веб-разработки.

11.6. Самостоятельное упражнение

Непосредственно при открытии страницы 11.2.html, написанной в *разд. 11.2*, активная гиперссылка не подсвечивается. Исправьте этот недочет.

Подсказка: реализуйте подсветку гиперссылки в обработчике события `load` окна, а значение якоря получайте из свойства `hash` объекта `location`.

ЧАСТЬ III

HTML API и компонентное программирование

- Таймеры и фоновые потоки.
- Работа с внешними данными.
- Перетаскивание.
- Холст и программная графика.
- Объявление своих классов.
- Компоненты.

Урок 12
Таймеры и фоновые потоки

Таймеры: периодические и однократные
Понятие HTML API
Фоновые потоки

12.1. Упражнение. Используем периодические таймеры

Напишем клиентское веб-приложение, вычисляющее простые числа в указанном пользователем диапазоне.

1. В папке 12\!sources сопровождающего книгу электронного архива (см. *приложение 3*) найдем страницу 12.1.1.html и скопируем ее куда-либо на локальный диск. Эта страница содержит заготовку для создания приложения (рис. 12.1).

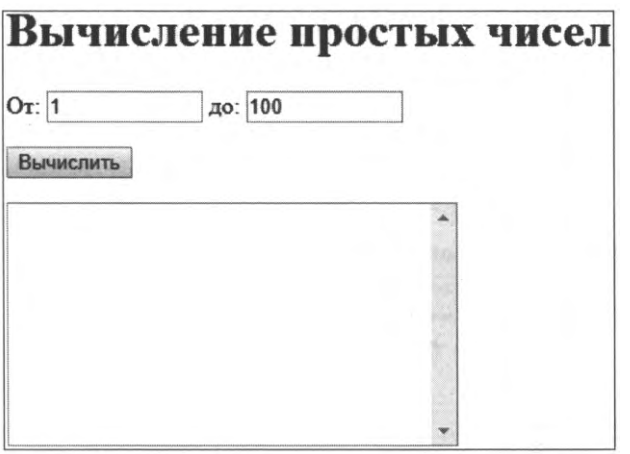


Рис. 12.1. Заготовка для создания клиентского приложения

2. Откроем страницу 12.1.1.html в текстовом редакторе и найдем код, создающий приведенную на рисунке веб-форму:

```
<form>
    <p>От: <input type="number" min="0" value="1">
    до: <input type="number" min="0" value="100"></p>
    <p><input type="submit" value="Вычислить"></p>
    <p><select size="10"></select></p>
</form>
```

В поля ввода чисел **От:** и **до:** заносятся начальное и конечное числа диапазона, в котором нужно найти простые числа, кнопка отправки данных **Вычислить** запускает вычисления, а нижний список, изначально пустой, служит для вывода найденных простых чисел.

3. В конце HTML-кода добавим тег `<script>` с первыми выражениями сценария:

```
<script type="text/javascript">
    const form = document.forms[0];
    form.addEventListener('submit', (evt) => {
        const min = parseInt(form.elements[0].value);
        const max = parseInt(form.elements[1].value);
        let flag, sqr, el;
        form.elements[3].innerHTML = '';
        form.elements[2].disabled = true;
    });
</script>
```

Все вычисления будут выполняться в обработчике события `submit` формы. В нем мы извлекаем заданные пользователем начальное и конечное числа диапазона, приводим их к числовому типу, очищаем нижний список и делаем недоступной кнопку **Вычислить**, дав пользователю понять, что в текущий момент идет поиск простых чисел.

4. Добавим в тело обработчика код, ищущий простые числа (добавленный код здесь и далее выделен полужирным шрифтом):

```
form.addEventListener('submit', (evt) => {
    . . .
    form.elements[2].disabled = true;
    for (let i = min; i <= max; ++i) {
        if (i % 2 != 0) {
            flag = true;
            sqr = Math.floor(Math.sqrt(i));
            for (let j = 3; j <= sqr; j += 2)
                if (i % j == 0) {
                    flag = false;
                    break;
                }
            if (flag) {
                el = document.createElement('option');
                el.textContent = i;
                form.elements[3].add(el);
            }
        }
    }
});
```

Принцип несложен: простым считается число, не делящееся нацело на 2 и все числа от трех до значения квадратного корня из этого числа, округленного до ближайшего меньшего целого. Такое число добавляется в нижний список.

5. Добавим в тело обработчика выражения, делающие кнопку **Вычислить** доступной и отменяющие обработчик события по умолчанию:

```
form.addEventListener('submit', (evt) => {
    . . .
    for (let i = min; i <= max; ++i) {
        . . .
    }
    form.elements[2].disabled = false;
    evt.preventDefault();
});
```

Откроем страницу 12.1.1.html в веб-обозревателе, оставим в полях **От:** и **до:** присутствующие там значения по умолчанию (`0` и `100` соответственно) и нажмем кнопку **Вычислить** — в нижнем списке появятся все простые числа, найденные в указанном диапазоне (рис. 12.2).

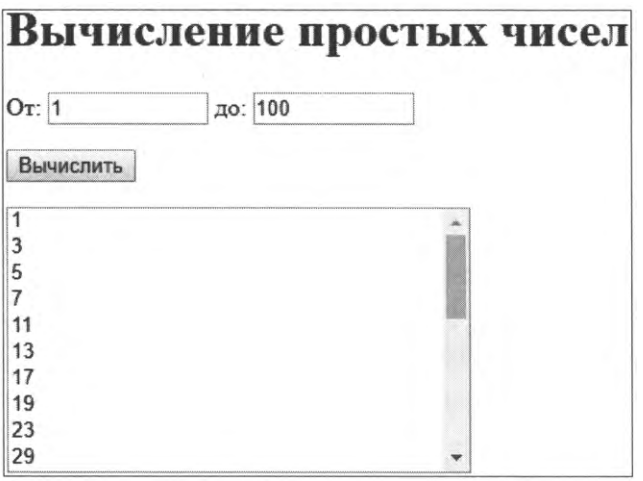


Рис. 12.2. В нижнем списке появились все простые числа, найденные в заданном диапазоне

Проблема здесь в том, что пока идут вычисления, приложение перестает отвечать на действия пользователя (как бы «подвисает») — поскольку веб-обозреватель, полностью занятый выполнением кода, не успевает отреагировать на действия пользователя вовремя. Невозможно даже прокрутить список, чтобы увидеть найденные простые числа.

Можно разбить процесс вычислений на отдельные короткие этапы, выполняемые через одинаковые промежутки времени. Тогда веб-обозреватель сможет отреагировать на действия пользователя в промежутке между выполнением отдельных этапов. Реализовать такой подход проще всего посредством периодических таймеров.

> *Периодический таймер* — программный механизм HTML API, многократно срабатывающий по истечении указанного промежутка времени и выполняющий при срабатывании заданную функцию.

> *HTML API* (HTML Application Programming Interface, интерфейс программирования приложений HTML) — набор программных интерфейсов, не относящихся к DOM и BOM. В их число входят средства для работы с таймерами и фоновыми потоками, рассматриваемые в этом уроке.

Создадим периодический таймер, который будет срабатывать каждые 0,1 с и вызывать функцию, обрабатывающую за один вызов 1000 чисел.

Периодический таймер создается методом `setInterval` класса `Window`:

```
setInterval(<функция>, <промежуток времени>↵
[, <параметры, передаваемые функции>])
```

Функция, вызываемая при срабатывании таймера, может быть именованной, анонимной или функцией-стрелкой. *Промежуток времени* указывается в миллисекундах. Последующими *параметрами* могут быть переданы значения, которые получит в качестве параметров функция.

Метод возвращает целочисленный идентификатор таймера, который в дальнейшем может быть использован для удаления этого таймера.

Ранее созданный периодический таймер удаляется методом `clearInterval` класса `Window`:

```
clearInterval(<идентификатор таймера>)
```

После удаления таймер перестанет срабатывать, и указанная в нем функция перестанет вызываться.

Итак, продолжим работу над нашим клиентским приложением.

1. Пересохраним страницу 12.1.1.html под именем 12.1.2.html, откроем ее в текстовом редакторе и удалим все содержимое тега `<script>`.
2. Запишем в тег `<script>` первые выражения нового сценария:

   ```
   const form = document.forms[0];
   let min, max, i, int;
   ```

   Мы объявили переменные `min` и `max` для хранения начального и конечного чисел диапазона, `i` — для хранения текущего обрабатываемого числа, `int` — под идентификатор таймера.
3. Объявим функцию `computeSimpleValues`, которая будет обрабатывать очередные 1000 чисел из заданного диапазона:

   ```
   function computeSimpleValues() {
       let flag, sqr, el;
       for (let k = 0; i <= max && k <= 1000; ++i, ++k) {
           if (i % 2 != 0) {
               flag = true;
               sqr = Math.floor(Math.sqrt(i));
               for (let j = 3; j <= sqr; j += 2)
                   if (i % j == 0) {
                       flag = false;
   ```

```
                    break;
                }
            if (flag) {
                el = document.createElement('option');
                el.textContent = i;
                form.elements[3].add(el);
            }
        }
    }
    if (i >= max) {
        window.clearInterval(int);
        form.elements[2].disabled = false;
    }
}
```

Код этой функции похож на код обработчика события `submit` из предыдущего варианта сценария, но с двумя различиями. Во-первых, цикл, в котором производится поиск простых чисел, будет выполняться, пока текущее число из переменной `i` не выходит за верхнюю границу (`max`), и количество обработанных чисел не превышает 1000. Во-вторых, по окончании обработки всех чисел из диапазона (если значение переменной `i`, которая в тот момент будет хранить последнее из обработанных чисел, выходит за верхнюю границу диапазона `max`) мы удаляем таймер вызовом метода `clearInterval` и делаем доступной кнопку **Вычислить**, тем самым приведя приложение в изначальное состояние.

4. Наконец, напишем обработчик события `submit`, запускающий процесс вычисления:

```
form.addEventListener('submit', (evt) => {
    min = parseInt(form.elements[0].value);
    max = parseInt(form.elements[1].value);
    i = min;
    form.elements[3].innerHTML = '';
    form.elements[2].disabled = true;
    int = window.setInterval(computeSimpleValues, 100);
    evt.preventDefault();
});
```

Не забываем занести начальную границу диапазона в переменную `i`.

Откроем страницу 12.1.2.html в веб-обозревателе и проверим, работает ли она.

12.2. Однократный таймер

> *Однократный таймер* — таймер, срабатывающий однократно и автоматически удаляющийся после этого.

Однократный таймер создается вызовом метода `setTimeout` класса `Window`, полностью аналогичного методу `setInterval` (см. *разд. 12.1*):

```
setTimeout(<функция>, <промежуток времени>↵
[, <параметры, передаваемые функции>])
```

Чтобы удалить однократный таймер до срабатывания, следует вызвать метод `clearTimeout` класса `Window`:

```
clearTimeout(<идентификатор таймера>)
```

12.3. Упражнение. Применяем фоновые потоки

Несмотря на все ухищрения, приложение, написанное в *разд. 12.1*, все равно временами «притормаживает». Перепишем его, вынеся процесс вычисления в фоновый поток, что повысит отзывчивость приложения.

> *Поток* — единица исполнения программного кода, выделяемая операционной системой. Пока поток занят выполнением одного сценария, другой сценарий он исполнить не сможет.

> *Основной поток* — поток, в котором выполняется код обычных веб-сценариев.

> *Фоновый поток* — поток, работающий параллельно с основным. Часто применяется для выполнения длительных вычислений, что позволяет сделать приложение отзывчивее для пользователей.

Чтобы запустить программный код в фоновом потоке, необходимо:

- вынести этот код в отдельный файл сценария;
- в основном коде создать объект класса `Worker`, применив оператор `new`:

```
new Worker(<ссылка на файл сценария с кодом, ↵
выносимым в отдельный поток>)
```

Объект фонового потока необходимо сохранить в какой-либо переменной.

> Фрагменты кода, выполняемые в разных потоках, совершенно изолированы друг от друга. Ни один поток не имеет доступ к переменным и функциям, объявленным в других потоках.

1. В папке 12\!sources сопровождающего книгу электронного архива (см. *приложение 3*) найдем файл 12.1.1.html и пересохраним его где-либо на локальном диске под именем 12.3.html.
2. Откроем файл 12.3.html в текстовом редакторе и добавим в конце HTML-кода тег `<script>` с первыми выражениями основного сценария:

```
<script type="text/javascript">
    const form = document.forms[0];
    let worker;
</script>
```

В переменной `worker` мы позднее сохраним объект фонового потока.

3. Начнем писать код обработчика события `submit` формы:

```
form.addEventListener('submit', (evt) => {
    form.elements[3].innerHTML = '';
    form.elements[2].disabled = true;
});
```

4. Теперь нужно создать фоновый поток. Условимся, что выполняющийся в нем код мы сохраним в файле 12.3.js.

   В обработчик события `submit` добавим выражение, создающее поток:

```
form.addEventListener('submit', (evt) => {
    . . .
    form.elements[2].disabled = true;
    worker = new Worker('12.3.js');
});
```

5. Поскольку все потоки независимы друг от друга, сценарий из фонового потока не сможет получить доступ к списку, чтобы добавить в него найденное простое число. Для пересылки чисел из фонового потока в основной мы применим межпоточные сообщения.

   > *Межпоточное сообщение* — сигнал, пересылаемый из одного потока другому и содержащий в себе одно произвольное значение. Применяется для обмена данными между потоками и пересылки управляющих команд.

   Чтобы получить в основном потоке сообщение, отправленное фоновым потоком, достаточно обработать событие `message`, возникающее в объекте фонового потока.

   Для этого добавим выражение, привязывающее к событию `message` потока `worker` функцию-обработчик `processResult` (будет написана позже):

```
form.addEventListener('submit', (evt) => {
    . . .
    worker = new Worker('12.3.js');
    worker.addEventListener('message', processResult);
});
```

6. Код из фонового потока требует вводные данные: начальное и конечное значения диапазона. Создадим служебный объект класса `Object` со свойствами `min` и `max`, сохраним в них эти данные и перешлем объект фоновому потоку в межпоточном сообщении.

   > Пересылка межпоточного сообщения от основного потока фоновому выполняется вызовом у объекта фонового потока метода `postMessage` в формате:
   >
   > `postMessage(<`*значение, пересылаемое в сообщении*`>)`.

   Добавим код, пересылающий вводные данные в фоновый поток:

```
form.addEventListener('submit', (evt) => {
    . . .
```

```
    worker.addEventListener('message', processResult);
    const data = {};
    data.min = parseInt(form.elements[0].value);
    data.max = parseInt(form.elements[1].value);
    worker.postMessage(data);
    evt.preventDefault();
});
```

7. Нам также нужно реагировать на сообщения от фонового потока и на хранящиеся в них значения. Условимся, что строка `'stop'` станет командой на завершение фонового потока, а любое другое значение будет выведено в списке найденных простых чисел.

> Для завершения фонового потока у его объекта следует вызвать не принимающий параметров метод `terminate`.

Перед обработчиком события `submit` запишем код объявления функции `processResult`, которая выполнит все это:

```
function processResult(evt) {
    if (evt.data == 'stop') {
        form.elements[2].disabled = false;
        worker.terminate();
    } else {
        const el = document.createElement('option');
        el.textContent = evt.data;
        form.elements[3].add(el);
    }
}
```

8. Для получения сообщений от основного потока в фоновом потоке обработчик события `message` следует привязать к объекту фонового потока, а для отправки сообщения из фонового потока основному — вызвать метод `postMessage` также у объекта фонового потока. Упомянутый ранее объект можно получить из переменной `this`, автоматически создаваемой в фоновом потоке.

Создадим в той же папке файл 12.3.js, в котором сохраним код фонового потока:

```
this.addEventListener('message', (evt) => {
    let flag, sqr;
    for (let i = evt.data.min; i <= evt.data.max; ++i)
        {
        if (i % 2 != 0) {
            flag = true;
            sqr = Math.floor(Math.sqrt(i));
            for (let j = 3; j <= sqr; j += 2)
                if (i % j == 0) {
                    flag = false;
                    break;
                }
```

```
            if (flag)
                this.postMessage(i);
        }
    }
    this.postMessage('stop');
});
```

Как только очередное простое число найдено, отправляем его в сообщении основному потоку для вывода в списке. По завершении вычислений не забываем отправить сообщение со строкой `'stop'` — командой на завершение фонового потока.

> ***Внимание!***
>
> В веб-обозревателе Google Chrome фоновые потоки могут запускаться лишь на страницах, загруженных с веб-сервера. Попытка запустить фоновый поток на странице, загруженной с локального диска, вызовет ошибку.

Откроем в любом веб-обозревателе, за исключением Chrome, страницу 12.3.html (автор тестировал ее в Mozilla Firefox) и проверим ее в работе.

12.4. Самостоятельные упражнения

- На страницу 12.1.2.html добавьте кнопку **Стоп**, при нажатии на которую вычисления будут прерываться. Сделайте так, чтобы в случае, если вычисления еще не запущены, эта кнопка была бы недоступна.
- Сделайте аналогичную кнопку на странице 12.3.html.

Урок 13
Работа с файлами, хранение данных и перетаскивание

Получение сведений о выбранных файлах
Чтение и вывод файлов
Хранение данных в локальном хранилище
Перетаскивание

13.1. Работа с локальными файлами

> Веб-обозреватель позволяет работать только с теми локальными файлами, которые были выбраны пользователем. Считывать произвольные файлы с локального диска он не может.

Выбор файлов для обработки в веб-сценарии или отправки серверу выполняется в поле выбора файлов. Оно формируется одинарным тегом `<input>`, в котором записывается атрибут `type` со значением `file`:

```
<input type="file" id="file" multiple>
```

Класс такого поля поддерживает свойство `files`. Оно хранит объект класса `FileList`, который является коллекцией выбранных в поле файлов (если в теге `<input>` отсутствует атрибут без значения `multiple`, коллекция будет содержать лишь один файл). Каждый из элементов коллекции — это объект класса `File`, представляющий отдельный выбранный файл.

13.1.1. Чтение сведений о файлах

В свойствах класса `File` (табл. 13.1) хранятся имя, тип, размер и дата последнего изменения файла.

***Таблица 13.1.** Свойства класса `File`*

Свойство	Описание
`name`	Имя файла без пути к нему
`type`	MIME-тип
`size`	Размер в байтах в виде числа
`lastModifiedDate`	Временная отметка последнего изменения файла в виде объекта класса `Date`

Вот пример считывания сведений о выбранных файлах:

```
<p><input type="file" id="file" multiple></p>
<div id="output"></div>
. . .
const file = document.getElementById('file');
const output = document.getElementById('output');
file.addEventListener('change', () => {
    let s = '', f;
    for (let i = 0; i < file.files.length; ++i) {
        f = file.files[i];
        s += '<h2>' + f.name + '</h2>';
        s += '<p>Размер: ' + f.size + ' байтов</p>';
        s += '<p>Тип: ' + f.type + '</p>';
        s += '<p>Последнее изменение: ' +
            f.lastModifiedDate.toLocaleString() + '</p>';
    }
    output.innerHTML = s;
});
```

13.1.2. Считывание текстовых файлов. Класс *FileReader*

Класс `FileReader` позволяет прочитать содержимое текстового файла, выбранного пользователем. Для этого необходимо:

- воспользовавшись оператором `new`, создать объект класса `FileReader` без указания параметров создаваемого объекта;
- привязать к созданному ранее объекту обработчик события `load`, который выполнится после загрузки файла и прочитает его содержимое;
- в теле обработчика извлечь прочитанное содержимое текстового файла из свойства `result` текущего объекта класса `FileReader`;
- вне обработчика запустить операцию считывания файла, вызвав у объекта класса `FileReader` метод `readAsText`:

  `readAsText(`*<считываемый файл>*`[, `*<кодировка>*`])`

  Считываемый файл указывается в виде объекта класса `File`. *Кодировка* задается в виде строки. Если она не указана, используется UTF-8.

Вот пример считывания выбранного пользователем текстового файла и вывода его содержимого на экран:

```
<p><input type="file" id="file" accept=".txt"></p>
<div id="output"></div>
. . .
const file = document.getElementById('file');
const output = document.getElementById('output');
const fr = new FileReader();
```

```
fr.addEventListener('load', (evt) => {
    output.textContent = evt.target.result;
});
file.addEventListener('change', () => {
    fr.readAsText(file.files[0]);
    // Чтобы считать файл в кодировке Windows-1251,
    // следует использовать выражение:
    // fr.readAsText(file.files[0], 'windows-1251');
});
```

Класс `FileReader` поддерживает ряд событий (табл. 13.2). Все события, за исключением `progress`, представляются классом `Event`. Свойства события `progress` представлены в табл. 13.3.

Таблица 13.2. *События класса* `FileReader`

Имя события	Когда возникает?
`loadstart`	Запуск считывания содержимого файла
`progress`	Периодически при считывании очередного фрагмента содержимого
`loadend`	Завершение считывания, неважно, успешном или нет
`load`	Успешное завершение считывания
`error`	Возникновение ошибки в процессе считывания
`abort`	Принудительное прерывание считывания

Таблица 13.3. *Свойства события* `progress`

Свойство	Хранимое значение
Класс `ProgressEvent` Представляет событие `progress`	
`lengthComputable`	`true`, если возможно вычислить размер выбранного файла, и `false` в противном случае
`loaded`	Размер загруженной части файла в байтах
`total`	Размер всего файла в байтах

Имеются также еще три свойства, доступные только для чтения:

- `result` — содержимое прочитанного файла. Действительно только в случае успешного считывания;
- `readyState` — состояние объекта в виде одного из чисел:
  - `0` — считывание еще не запущено;
  - `1` — считывание идет;
  - `2` — содержимое файла полностью загружено;

- `error` — объект исключения, сгенерированного в случае возникновения ошибки при считывании файла.

13.1.3. Вывод индикатора процесса при считывании файла

Для вывода индикатора, показывающего процесс считывания файла, используется объект события `progress`, относящийся к классу `ProgressEvent`:

```
<p>Загружено <span id="prg"></span></p>
. . .
const prg = document.getElementById('prg');
const fr = new FileReader();
. . .
fr.addEventListener('progress', (evt) => {
    if (evt.lengthComputable)
        prg.textContent = evt.loaded / evt.total * 100 + '%';
});
```

13.1.4. Считывание графических файлов

Для считывания графических файлов с целью вывода применяется тот же подход и тот же класс `FileReader`. Есть лишь два различия:

- загрузка запускается вызовом метода `readAsDataURL`:

  `readAsDataURL(<`*`загружаемый файл`*`>)`

- в свойстве `result` объекта, выполняющего загрузку файла, окажется Data URL с содержимым этого файла.

> *Data URL* — интернет-адрес, представляющий собой закодированное содержимое какого-либо файла. Может быть использован так же, как и обычный интернет-адрес (в атрибуте `src` тега `<img>` и др.).

13.2. Упражнение. Реализуем предварительный просмотр выбранного графического файла

Часто в веб-формах для выгрузки графического файла на сервер делают панель предварительного просмотра. Давайте и мы сделаем нечто подобное.

1. В папке 15\!sources сопровождающего книгу электронного архива (см. *приложение 3*) найдем страницу 13.2.html и скопируем ее куда-либо на локальный диск.

   Эта страница содержит поле выбора файла `file`, расположенный под ней блок `preview`, в котором будет выводиться выбранное изображение, и оформляющую все это внутреннюю таблицу стилей.

2. Откроем копию страницы 13.2.html в текстовом редакторе и вставим под HTML-кодом тег `<script>` с первыми выражениями сценария, которые получат доступ к полю выбора файла, блоку и создадут объект класса `FileReader`:

```
<script type="text/javascript">
    const file = document.getElementById('file');
    const preview = document.getElementById('preview');
    const fr = new FileReader();
</script>
```

3. Добавим обработчик события `load` только что созданного объекта, который выведет загруженный файл в блоке `preview`:

```
fr.addEventListener('load', (evt) => {
    preview.style.backgroundImage = 'url(' + evt.target.result + ')';
});
```

Мы выведем загруженное изображение в виде графического фона, поскольку такой фон проще разместить по центру блока.

4. Добавим обработчик события `change` поля выбора файла `file`, который запустит загрузку выбранного файла:

```
file.addEventListener('change', () => {
    fr.readAsDataURL(file.files[0]);
});
```

Откроем страницу 13.2.html в веб-обозревателе, выберем в поле какой-либо графический файл и посмотрим на его содержимое, выведенное в блоке (рис. 13.1).

Рис. 13.1. Вывод на экран изображения из выбранного графического файла

13.3. Хранение данных на стороне клиента

В хранилище веб-обозревателя можно временно сохранить любые данные (например, сведения, занесенные пользователем в форму).

> *Хранилище* — область памяти, предназначенная для временного сохранения произвольных данных на стороне клиента.

Для всех веб-страниц, загруженных с одного хоста, выделяется отдельное независимое хранилище с неограниченным объемом.

Доступ к хранилищу можно получить через два свойства класса `Window`:

- `sessionStorage` — *сессионное хранилище*, хранит данные, пока открыто текущее окно веб-обозревателя, а при его закрытии очищается. Подходит для временного сохранения данных на случай внезапной перезагрузки страницы и для передачи данных между страницами;
- `localStorage` — *постоянное хранилище*, хранит данные сколь угодно долго (пока те не будут явно удалены). Применяется для записи пользовательских настроек, сведений, предназначенных для позднейшей отправки на сервер, и т. п.

Оба хранилища представляются объектами класса `Storage` и, соответственно, имеют сходную функциональность.

Работа с хранилищами выполняется вызовами четырех методов класса `Storage`:

- `setItem(<имя значения>, <само значение>)` — записывает в хранилище *значение* под указанным *именем*. Если значение с таким *именем* уже имеется в хранилище, обновляет его;
- `getItem(<имя значения>)` — возвращает сохраненное ранее значение с указанным *именем* или `null`, если его в хранилище нет;
- `removeItem(<имя значения>)` — удаляет из хранилища значение с указанным *именем*;
- `clear()` — очищает хранилище.

> Все значения в хранилище хранятся в виде строк.

Пример:

```
// Считываем из хранилища значение с именем someValue
let someValue = localStorage.getItem('someValue');
if (someValue)
    // Если оно существует, преобразуем его в числовой тип
    someValue = parseInt(someValue);
else
    // Если такого значения в хранилище нет, задаем ему
    // величину по умолчанию
    someValue = 10;
```

```
// Сохраняем значение someValue
localStorage.setItem('someValue', someValue);
```

13.4. Перетаскивание

> *Перетаскивание* (иначе *drag'n'drop*) — технология, позволяющая переносить данные из одного элемента страницы (*источника*) в другой (*приемник*) простым перетаскиванием мышью.

Реализация перетаскивания на странице включает несколько этапов, которые мы рассмотрим в порядке их выполнения.

13.4.1. Превращение элемента-источника в перетаскиваемый

Можно перетаскивать элемент страницы мышью или нет, зависит от значения атрибута `draggable`, поддерживаемого всеми тегами:

- `"auto"` — можно перетаскивать только изображения (`<img>`) и гиперссылки (`<a>`). Это поведение по умолчанию;
- `"true"` — перетаскивать можно;
- `"false"` — перетаскивать нельзя.

Пример:

```
<p draggable="true">Тащите меня семеро!</p>
```

13.4.2. Задание перемещаемых данных в источнике

Целью перетаскивания является перенос данных из источника в приемник. Переносимые данные задаются в источнике в обработчике любого из следующих событий:

- `dragstart` — возникает, когда пользователь начинает операцию перетаскивания;
- `drag` — возникает, когда операция перетаскивания реально начинается.

Эти события всплывающие, имеют обработчик по умолчанию, запускающий операцию перетаскивания (если обработчик отменить, она не начнется).

Оба события представляются классом `DragEvent`. Он поддерживает свойство `dataTransfer` с объектом класса `DataTransfer`, хранящим переносимые данные. Один такой объект может содержать целый набор переносимых значений, относящихся к разным типам (например, строку обычного текста, фрагмент HTML-кода и интернет-адрес).

Для занесения нового значения в набор переносимых данных применяется метод `setData` класса `DataTransfer`:

```
setData(<тип значения>, <само значение>)
```

Оба параметра указываются в виде строк. *Тип значения* можно задать произвольный, но обычно используют MIME-тип.

Метод `clearData` того же класса удаляет из набора ненужное значение:

```
clearData([<тип значения>])
```

Если *тип значения* не указан, из набора будут удалены все значения.

Пример:

```
<p id="dragsource" draggable="true">Тащите меня семеро!</p>
. . .
const dragsource = document.getElementById('dragsource');
dragsource.addEventListener('dragstart', (evt) => {
    evt.dataTransfer.setData('text/plain',
        'Важные данные. Тащите аккуратнее.');
    evt.dataTransfer.setData('text/html',
        'Важные данные. <em>Тащите аккуратнее.</em>');
});
```

13.4.3. Указание допустимых операций

В процессе перетаскивания пользователь может выбрать одну из трех операций:

- *перемещение данных* — при простом перетаскивании;
- *копирование данных* — при удержании клавиши <Ctrl>;
- *создание ссылки на данные* — при удержании клавиши <Alt> (реализуется крайне редко).

Разработчик сценария может ограничить набор допустимых операций, задав его в свойстве `effectAllowed` класса `DataTransfer`, поддерживающем значения:

- `'all'` — допустимы все три операции;
- `undefined` — то же, что и `'all'` (значение по умолчанию);
- `'move'` — допускается только перенос данных;
- `'copy'` — только копирование;
- `'link'` — только создание ссылки;
- `'copyMove'` — перенос и копирование;
- `'linkMove'` — перенос и создание ссылки;
- `'copyLink'` — копирование и создание ссылки;
- `'none'` — не допускается ни одна операция.

Эти свойства указываются там же, где выполняется занесение данных, — в обработчике события `dragstart` или `drag` источника:

```
dragsource.addEventListener('dragstart', (evt) => {
    . . .
    // Разрешаем только перенос и копирование данных
    evt.dataTransfer.effectAllowed = 'copyMove';
});
```

13.4.4. Подготовка элемента-приемника

Чтобы превратить элемент страницы в приемник для перетаскивания, достаточно:

- привязать обработчик к событию `dragover`, периодически возникающему в приемнике при перемещении курсора мыши над ним;
- в обработчике отменить обработку события по умолчанию (которая блокирует перетаскивание) вызовом метода `preventDefault` у объекта события.

Тогда при наведении на приемник курсор мыши изменит форму, обозначая выбранную пользователем операцию: перемещение, копирование или создание ссылки.

Пример:

```
<div id="dragdest">Валите все сюда!</div>
. . .
const dragdest = document.getElementById('dragdest');
dragdest.addEventListener('dragover', (evt) => {
    evt.preventDefault();
});
```

Если в процессе перетаскивания всегда выполняется какая-либо одна операция из упомянутых, можно указать ее принудительно. Для этого применяется свойство `dropEffect` класса `DataTransfer`, поддерживающее значения:

- `'move'` — перемещение данных;
- `'copy'` — копирование;
- `'link'` — создание ссылки;
- `'none'` — любые операции запрещены.

Пример:

```
dragdest.addEventListener('dragover', (evt) => {
    // Всегда будет выполняться перенос данных
    evt.dataTransfer.dropEffect = 'move';
    evt.preventDefault();
});
```

В приемнике также возникают следующие события:

- `dragenter` — возникает при наведении курсора мыши на элемент (обработчик по умолчанию, как и у события `dragover`, не дает завершить перетаскивание);
- `dragleave` — возникает при уводе курсора мыши с элемента (обработчик по умолчанию отсутствует).

В обработчиках этих событий можно, например, визуально выделять элемент-приемник с целью показать, что в него можно «положить» переносимые данные.

Все три рассмотренных здесь события: `dragover`, `dragenter` и `dragleave` — всплывающие.

13.4.5. Завершение перетаскивания

Когда пользователь отпускает кнопку мыши над приемником, в последнем возникает событие `drop`. Код, завершающий операцию переноса данных, записывается в обработчике этого события.

Набор переносимых значений находится в объекте класса `DataTransfer`, хранящемся в свойстве `dataTransfer` объекта события. Извлечь их поможет метод `getData` класса `DataTransfer`:

```
getData(<тип извлекаемого значения>)
```

Он возвращает хранящееся в текущем наборе значение, относящееся к указанному *типу*, в виде строки. Если в наборе отсутствует значение такого *типа*, возвращается пустая строка.

Событие `drop` всплывающее, имеет обработчик по умолчанию, который обрабатывает операцию перетаскивания самостоятельно.

> ***Внимание!***
>
> Практически всегда обработчик события `drop` по умолчанию принимает переносимые данные за интернет-адрес и пытается выполнить переход по нему. Поэтому обработку события `drop` по умолчанию следует отменять.

Пример:

```
dragdest.addEventListener('drop', (evt) => {
    let s = evt.dataTransfer.getData('text/plain');
    if (evt.dataTransfer.dropEffect == 'move')
        // Выполняем перенос данных
    else
        // Копируем данные
    evt.preventDefault();
});
```

По завершении перетаскивания в элементе-источнике возникает событие `dragend`. Оно всплывающее, не имеет обработчика по умолчанию и может применяться для выдачи каком-либо оповещений о завершении переноса данных.

13.5. Упражнение. Практикуемся в реализации перетаскивания

Напишем веб-приложение «Экзаменатор», предлагающее перенести из левого перечня в правый все языки для разработки веб-страниц (но не серверных веб-приложений!). Для этого мы применим знания, полученные в *разд. 13.4*.

1. В папке 13\!sources сопровождающего книгу электронного архива (см. *приложение 3*) найдем файл 13.5.html и скопируем его куда-либо на локальный диск. Этот файл хранит страницу с заготовкой для написания веб-приложения (рис. 13.2).

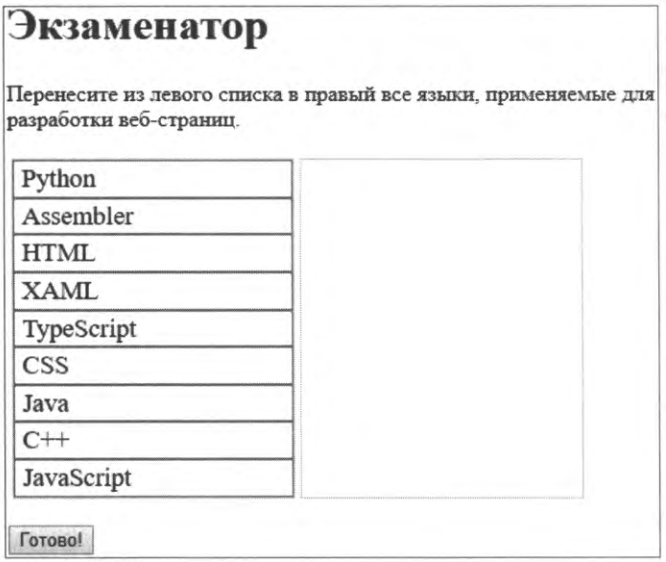


Рис. 13.2. Заготовка для написания веб-приложения

Оба перечня созданы обычными блоками, отдельные позиции этих перечней — также блоками. Левый перечень имеет якорь `all`, правый — `selected`, кнопка **Готово** — `done`, а блок для вывода результатов — `output`.

2. Откроем копию страницы 13.5.html в текстовом редакторе и поместим под HTML-кодом тег `<script>` с первыми выражениями сценария, которые получат доступ к нужным элементам страницы:

```
<script type="text/javascript">
    const all = document.getElementById('all');
    const selected = document.getElementById('selected');
</script>
```

3. Перенос данных инициируем в обработчике события `dragstart`, который привяжем к отдельным позициям перечня `all`. В обработчике занесем в набор переносимых данных якорь перетаскиваемого элемента (который сформируем программно чуть позже) и разрешим только операцию перемещения.

   Напишем обработчик события `dragstart`:

```
function dragStart(evt) {
    evt.dataTransfer.setData('text/id', this.id);
    evt.dataTransfer.effectAllowed = 'move';
}
```

   В вызове метода `setData` можно использовать существующий MIME-тип данных, а можно придумать свой, точнее описывающий передаваемые данные. Мы так и поступили, придумав тип `text/id`, обозначающий якорь.

4. Добавим код, привязывающий обработчик события к позициям перечня `all` и выполняющий другие необходимые действия:

```
for (let i = 0; i < all.children.length; ++i) {
    all.children[i].addEventListener('dragstart', dragStart);
    all.children[i].draggable = 'true';
    all.children[i].id = 'all' + i;
}
```

В число этих действий входит разрешение перетаскивания позиций и указание у них якорей вида `all<порядковый номер>`.

5. Добавим код обработчика события `dragover` перечня `selected`, в котором разрешим перемещение данных:

```
function dragOver(evt) {
    evt.dataTransfer.dropEffect = 'move';
    evt.preventDefault();
}
selected.addEventListener('dragover', dragOver);
```

6. Напишем обработчик события `drop` того же перечня, который извлечет переданный ему якорь, найдет по нему перемещаемую позицию перечня и выполнит ее перемещение в перечень `selected`:

```
function drop(evt) {
    const s = evt.dataTransfer.getData('text/id');
    if (s) {
        const el = all.querySelector('#' + s);
        selected.appendChild(el);
        evt.preventDefault();
    }
}
selected.addEventListener('drop', drop);
```

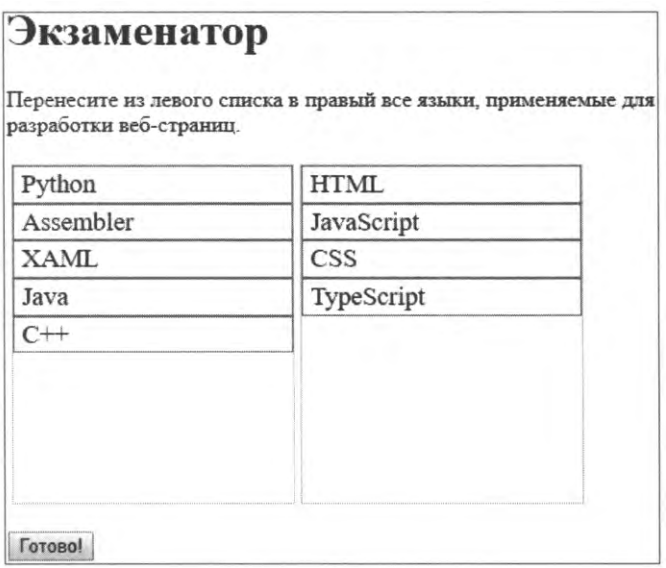


Рис. 13.3. Несколько позиций перетащены из левого перечня в правый

Не забываем отменить обработчик события `drop` по умолчанию, иначе веб-обозреватель отправит нас неизвестно куда.

Откроем страницу 13.5.html в веб-обозревателе и перетащим несколько позиций из левого перечня в правый (рис. 13.3).

13.6. Перетаскивание файлов в поле выбора файлов

Чтобы реализовать перетаскивание файлов из Проводника в поле выбора файлов, следует:

1. В обработчике события `dragover` поля выбора файлов разрешить перетаскивание, отменив обработку по умолчанию.
2. В обработчике события `drop` поля выбора файлов извлечь коллекцию полученных файлов из свойства `files` объекта набора данных. Эта коллекция представляет собой объект класса `FileList`, знакомый нам по *разд. 13.1*.

Пример:

```
<p><input type="file" id="file" multiple></p>
. . .
const file = document.getElementById('file');
file.addEventListener('dragover', (evt) => {
    evt.preventDefault();
});
file.addEventListener('drop', (evt) => {
    for (let i = 0; i < evt.dataTransfer.files.length; ++i) {
        // Перебираем и обрабатываем полученные файлы
    }
    evt.preventDefault();
});
```

13.7. Самостоятельные упражнения

- Напишите веб-приложение 13.7.html — просмотровщик выбранных пользователем файлов. Заготовку для него вы найдете в папке 13\!sources сопровождающего книгу электронного архива (см. *приложение 3*).

  Подсказки: для каждого файла создайте отдельный объект класса `FileReader`, по окончании загрузки файла создайте для него отдельный блок (`<div>`), выведите в нем изображение из файла в виде графического фона и добавьте блок в элемент с якорем `preview`.

  У вас должно получиться так, как показано на рис. 13.4.

- Реализуйте в приложении 13.7.html перетаскивание файлов из Проводника в поле выбора файлов.

Рис. 13.4. Просмотровщик выбранных пользователем файлов

- В приложении-экзаменаторе 13.5.html реализуйте перетаскивание позиций в обратном направлении — из правого списка в левый.
- Там же сделайте проверку правильности ответа по нажатию кнопки **Готово!**. В случае верного ответа в элементе `output` должна выводиться зеленая надпись «Правильно!», в случае неверного ответа — красная надпись «Неправильно!».

  Подсказка: для вывода надписи нужным цветом воспользуйтесь стилевыми классами `right` и `wrong`, уже имеющимися во внутренней таблице стилей.

Урок 14
Программная графика

Холст HTML
Рисование фигур
Вывод текста
Градиентные и графические заливки
Преобразования
Композиция

14.1. Холст HTML

В состав HTML API входят инструменты для программного рисования на страницах произвольной графики.

> *Холст* — встроенный элемент страницы, предназначенный для программного вывода графики.

Холст создается парным тегом `<canvas>` без содержимого. Поддерживаются два необязательных атрибута:

- `width` — ширина холста в пикселах (по умолчанию — 300);
- `height` — высота холста в пикселах (по умолчанию — 150).

Пример:

```
<canvas id="draw" width="640" height="480"></canvas>
```

Холст представляется классом `HTMLCanvasElement`. Он поддерживает свойства `width` и `height`, соответствующие одноименным атрибутам тега `<canvas>`.

> *Графический контекст* — объект, представляющий «поверхность» холста и, собственно, рисующий графику.

Для получения графического контекста следует вызвать у объекта холста метод `getContext`, передав ему в качестве параметра строку `'2d'`. Метод вернет в качестве результата объект класса `CanvasRenderingContext2D`, представляющий графический контекст:

```
const draw = document.getElementById('draw');
const ctx = draw.getContext('2d');
```

Рисование выполняется вызовом методов класса `CanvasRenderingContext2D`, которые будут рассматриваться далее.

Координаты на холсте и размеры рисуемых фигур указываются в пикселах. Начало координат находится в левом верхнем углу холста, а их значения отсчитываются в направлении вправо и вниз.

14.2. Рисование прямоугольников

Прямоугольники нарисовать проще всего — достаточно вызвать один из трех методов:

- `strokeRect(<x>, <y>, <ширина>, <высота>)` — рисует прямоугольник с левым верхним углом в точке с координатами [*x*, *y*] и с указанными *шириной* и *высотой*. Рисуемый прямоугольник представляет собой один контур без заливки;
- `fillRect(<x>, <y>, <ширина>, <высота>)` — то же самое, что и `strokeRect`, но выводимый прямоугольник содержит только заливку без контура.

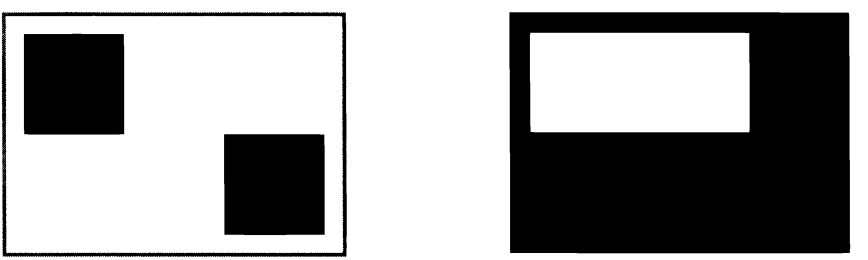
  Пример (рис. 14.1, *а*):

```
ctx1.strokeRect(10, 10, 170, 120);
ctx1.fillRect(20, 20, 50, 50);
ctx1.fillRect(120, 70, 50, 50);
```

- `clearRect(<x>, <y>, <ширина>, <высота>)` — очищает прямоугольную область с левым верхним углом в точке с координатами [*x*, *y*] и с указанными *шириной* и *высотой*.

  Пример (рис. 14.1, *б*):

```
ctx2.fillRect(10, 10, 170, 120);
ctx2.clearRect(20, 20, 110, 50);
```



а *б*

Рис. 14.1. *а* — выводимые прямоугольники содержат только заливку без контура; *б* — прямоугольная область залита черным цветом методом `fillRect`, и в ней очищена прямоугольная область методом `clearRect`

14.3. Указание основных параметров графики

Основные параметры графики, включающие цвета линий, заливок и настройки тени, действуют на любую графику, рисуемую на холсте.

14.3.1. Цвета линий и заливок

Для указания цвета рисуемой графики применяются свойства:

- `strokeStyle` — цвет линий контура;
- `fillStyle` — цвет заливок.

Значение цвета может быть указано в любом формате, поддерживаемым CSS. По умолчанию и контуры, и заливки выводятся черным цветом.

Пример (рис. 14.2):

```
ctx3.fillRect(10, 10, 170, 120);
ctx3.strokeStyle = '#ffffff';
ctx3.strokeRect(20, 20, 150, 100);
ctx3.strokeRect(40, 40, 110, 60);
ctx3.fillStyle = 'lightgrey';
ctx3.fillRect(60, 60, 70, 20);
```

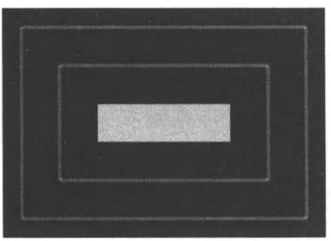


Рис. 14.2. Контуры и заливки выведены цветами в соответствии с приведенным примером кода

14.3.2. Параметры тени

Параметры тени, которая автоматически создается у рисуемых фигур, указываются посредством четырех свойств:

- `shadowBlur` — величина размытия в виде положительного числа с плавающей точкой или `0` (по умолчанию — `0`);
- `shadowColor` — цвет тени в любом формате, поддерживаемым CSS (по умолчанию — полностью прозрачный черный);
- `shadowOffsetX` — горизонтальное смещение тени в виде числа с плавающей точкой. Положительные значения вызывают смещение вправо, отрицательные — влево. Значение по умолчанию — `0`;
- `shadowOffsetY` — вертикальное смещение тени, задаваемое числом с плавающей точкой. Положительные значения вызывают смещение вниз, отрицательные — вверх. Значение по умолчанию — `0`.

Для вывода тени следует задать ненулевое значение, по крайней мере, у одного из свойств: `shadowBlur`, `shadowOffsetX` или `shadowOffsetY`.

Пример (рис. 14.3):

```
ctx2.strokeRect(10, 20, 30, 100);
ctx2.shadowColor = 'black';
ctx2.shadowBlur = 5;
ctx2.strokeRect(65, 20, 30, 100);
ctx2.shadowOffsetX = 15;
ctx2.shadowOffsetY = -10;
ctx2.strokeRect(120, 20, 30, 100);
```

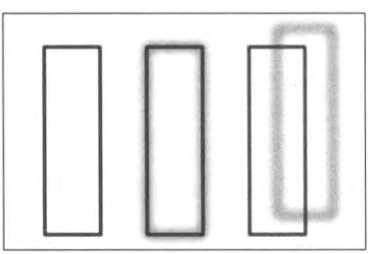


Рис. 14.3. Варианты задания теней в соответствии с приведенным примером кода

14.4. Вывод текста

Для вывода текста класс графического контекста предлагает такие методы:

- `strokeText(<текст>, <x>, <y>[, <максимальная ширина>])` — выводит указанный *текст* на экран. Левый конец базовой линии текста располагается в точке с координатами [*x*, *y*]. Если указана *максимальная ширина*, а ширина выводимого *текста* превышает ее, *текст* будет либо уменьшен в размерах (в случае незначительного превышения), либо выведен меньшим кеглем. Выведенный текст представляет собой один контур без заливки;
- `fillText(<текст>, <x>, <y>[, <максимальная ширина>])` — то же самое, что и `strokeText`, но выведенный текст содержит одну заливку без контура;
- `measureText(<текст>)` — возвращает объект класса `TextMetrics`, содержащий экранные размеры указанного *текста*. Класс `TextMetrics` поддерживает только свойство `width`, в котором записана ширина текста в пикселах.

Параметры выводимого текста задаются в следующих свойствах графического контекста:

- `font` — основные параметры текста (включая его гарнитуру, кегль, насыщенность и начертание) в формате, поддерживаемом атрибутом стиля CSS `font` (по умолчанию: `'10px sans-serif'`, т. е. шрифт без засечек кеглем 10 пикселов);
- `textAlign` — горизонтальное выравнивание текста относительно точки, указанной в вызове метода `strokeText` или `fillText`, в виде одной из строк:
  - `left` — по левому краю,
  - `center` — по центру;
  - `right` — по правому краю;
  - `start` — по левому краю для языков с направлением письма слева направо, по правому краю для языков с обратным направлением письма (значение по умолчанию);
  - `end` — по правому краю для языков с направлением письма слева направо, по левому краю для языков с обратным направлением письма;
- `textBaseline` — вертикальное выравнивание текста относительно базовой линии в виде одной из строк (рис. 14.4):
  - `alphabetic` — текст располагается на базовой линии (значение по умолчанию);
  - `top` — под базовой линией;

Рис. 14.4. Варианты вертикального выравнивания текста относительно базовой линии

- `middle` — середина текста совпадает с базовой линией;
- `bottom` — над базовой линией.

14.5. Упражнение. Рисуем диаграмму

Нарисуем на странице столбчатую диаграмму популярности различных языков программирования.

1. В папке 14\!sources сопровождающего книгу электронного архива (см. *приложение 3*) найдем файлы 14.5.html (с заготовкой страницы) и 14.5.js (с исходными данными для построения диаграммы). Скопируем их куда-либо на локальный диск.
2. Откроем копию файла 14.5.js в текстовом редакторе и посмотрим на его содержимое:

```
languages = [
    ['JavaScript', 70],
    ['Java', 45],
    ['Python', 39],
    ['C#', 34],
    ['PHP', 31],
    ['C++', 25]
];
```

   Исходные данные собраны в массив `languages`. Каждый его элемент содержит сведения об одном языке и также является массивом из двух элементов: названия языка и уровня его популярности в процентах.

3. Откроем копию страницы 14.5.html в текстовом редакторе и найдем код, создающий холст:

```
<canvas id="draw" width="500" height="200"></canvas>
```

   Впоследствии мы подгоним ширину холста так, чтобы построенная диаграмма поместилась в нем.

4. В секцию заголовка страницы 14.5.html вставим тег `<script>` (выделен полужирным шрифтом), привязывающий файл веб-сценария 14.5.js:

```
<head>
    . . .
    <script type="text/javascript" src="14.5.js"></script>
</head>
```

5. После HTML-кода поместим тег `<script>` с выражениями, получающими доступ к холсту и к его графическому контексту:

```
<script type="text/javascript">
    const draw = document.getElementById('draw');
    const ctx = draw.getContext('2d');
</script>
```

6. Чтобы вычислить ширину диаграммы, нужно знать ширину отдельного столбца. Она будет равна ширине самого длинного названия языка программирования. Условимся, что для вывода названий применим шрифт sans-serif кеглем 20 пикселов.

   Добавим в сценарий код, указывающий параметры шрифта и вычисляющий ширину столбцов:

   ```
   ctx.font = '20px sans-serif';
   let iwidth = 0, a;
   for (let i = 0; i < languages.length; ++i) {
       a = ctx.measureText(languages[i][0]).width;
       if (a > iwidth)
           iwidth = a;
   }
   ```

   После выполнения этого кода искомая ширина столбцов окажется в переменной `iwidth`.

7. Ширина диаграммы равна сумме совокупной ширины столбцов и совокупной ширине просветов между ними. Положим каждый такой просвет равным 5 пикселов.

   Добавим выражение, устанавливающее ширину холста равной ширине диаграммы:

   ```
   draw.width = iwidth * languages.length + 5 * (languages.length - 1);
   ```

   > После изменения размеров холста все настройки графики будут сброшены, а уже нарисованные фигуры — удалены.

8. Добавим выражения, вновь задающие параметры шрифта:

   ```
   ctx.font = '20px sans-serif';
   ctx.textAlign = 'center';
   ctx.textBaseline = 'bottom';
   ```

9. Добавим выражение, вычисляющее высоту самого высокого столбца:

   ```
   const iheight = draw.height - 25;
   ```

   Она равна высоте холста за вычетом 25 пикселов — под вывод названия языка.

10. И напишем код, выводящий диаграмму:

    ```
    let xi, yi;
    for (let i = 0; i < languages.length; ++i) {
        a = iheight * languages[i][1] / 100;
        xi = i * (iwidth + 5);
        yi = draw.height - a;
        ctx.fillRect(xi, yi, iwidth, a);
        ctx.fillText(languages[i][0], xi + iwidth / 2, yi, iwidth);
    }
    ```

При выводе надписи горизонтальную координату точки, в которой она будет выведена, устанавливаем в середину столбца — поскольку ранее указали для выводимого текста выравнивание по центру.

Откроем страницу 14.5.html в веб-обозревателе и посмотрим на нарисованную диаграмму (рис. 14.5).

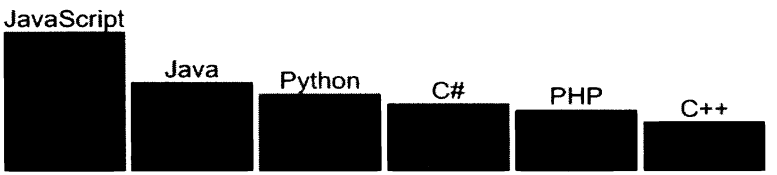


Рис. 14.5. Столбчатая диаграмма популярности различных языков программирования построена

14.6. Рисование сложных фигур

Для рисования более сложных, нежели прямоугольники, фигур графический контекст холста предлагает ряд следующих инструментов.

14.6.1. Начало и завершение рисования

Код, рисующий сложную фигуру, должен быть заключен в вызов двух методов, первый из которых запустит процесс рисования, а второй завершит его и, собственно, выведет фигуру на экран. Вот эти методы:

- `beginPath()` — запуск рисования;
- `stroke()` — завершение рисования с выводом только контуров нарисованных фигур:

```
ctx.beginPath();
// Рисуем контуры без заливки
ctx.stroke();
```

- `fill()` — завершение рисования с выводом только заливок нарисованных фигур:

```
ctx.beginPath();
// Рисуем фигуры из одних заливок без контуров
ctx.fill();
```

14.6.2. Перемещение пера

> *Перо* — воображаемый инструмент, рисующий сложные фигуры. Рисование фигуры начнется в точке, где в текущий момент находится перо, а по завершении перо окажется в конечной точке фигуры.

Перемещение пера в точку с координатами [*x*, *y*] выполняет метод `moveTo`:

```
moveTo(<x>, <y>)
```

Изначально перо установлено в точку [0, 0] — в левый верхний угол холста.

Рисование прямых линий

Прямую линию из точки, где находится перо, в точку с координатами [*x*, *y*] проводит метод `lineTo`:

```
lineTo(<x>, <y>)
```

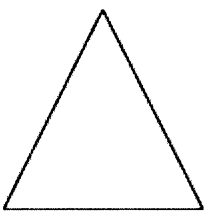
После рисования перо остается в точке [*x*, *y*].

Пример (рис. 14.6, *а*):

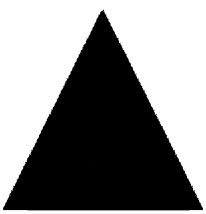
```
ctx1.beginPath();
ctx1.strokeStyle = 'blue';
ctx1.moveTo(50, 0);
ctx1.lineTo(100, 100);
ctx1.lineTo(0, 100);
ctx1.lineTo(50, 0);
ctx1.stroke();
```

Пример (рис. 14.6, *б*):

```
ctx2.beginPath();
ctx2.fillStyle = 'blue';
ctx2.moveTo(50, 0);
ctx2.lineTo(100, 100);
ctx2.lineTo(0, 100);
ctx2.lineTo(50, 0);
ctx2.fill();
```



а



б

Рис. 14.6. Рисование прямых линий: *а* — с выводом только контура нарисованной фигуры; *б* — с выводом только заливки нарисованной фигуры

Замыкание контура

Чтобы замкнуть рисуемый контур, проведя прямую линию от текущего местоположения пера до точки, в которой началось рисование, следует вызвать не принимающий параметров метод `closePath`.

Пример (см. рис. 14.6, *а*):

```
ctx3.beginPath();
ctx3.strokeStyle = 'blue';
ctx3.moveTo(50, 0);
ctx3.lineTo(100, 100);
ctx3.lineTo(0, 100);
ctx3.closePath();
ctx3.stroke();
```

При рисовании фигур из заливок методом `fill` контур будет замкнут автоматически (см. рис. 14.6, *б*).

Указание параметров линий

Мы можем указать у линий сложных фигур различные параметры: толщину, форму концов и пр.

> Эти параметры могут быть заданы в любом месте кода, расположенного между вызовами методов `beginPath` и `stroke/fill`. Они будут действовать на все линии рисуемой фигуры.

Для указания параметров применяются следующие свойства:

- `lineWidth` — толщина линий в пикселах (значение по умолчанию — `1`):

```
ctx1.beginPath();
ctx1.moveTo(0, 50);
ctx1.lineTo(80, 50);
ctx1.lineWidth = 5;
ctx1.stroke();
```

- `lineCap` — форма концов линий в виде строки:
  - `butt` — плоский (по умолчанию);
  - `round` — скругленный;
  - `square` — квадратный.

Пример:

```
ctx2.beginPath();
ctx2.moveTo(0, 50);
ctx2.lineTo(80, 50);
ctx2.lineWidth = 30;
ctx2.lineCap = 'round';
ctx2.stroke();
```

- `lineJoin` — форма соединений линий в виде строки:
  - `miter` — заостренная (по умолчанию);
  - `round` — скругленная;
  - `bevel` — срезанная.

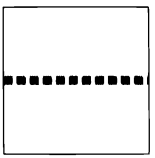
Помимо этого, можно указать стиль линий — будут ли они штриховыми или пунктирными. Для этого применяется метод `setLineDash` графического контекста:

`setLineDash(<`*массив с параметрами стиля линий*`>)`

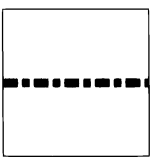
В передаваемом методу *массиве* все элементы должны представлять собой числа, которые поочередно указывают длину то очередного штриха, то очередного свободного пространства между двумя штрихами.

Примеры:

```
ctx3.beginPath();
ctx3.setLineDash([6, 3]);
ctx3.moveTo(0, 50);
ctx3.lineTo(100, 50);
ctx3.stroke();
```



```
ctx4.beginPath();
ctx4.setLineDash([10, 3, 5, 3]);
ctx4.moveTo(0, 50);
ctx4.lineTo(100, 50);
ctx4.stroke();
```



Чтобы вновь проводить сплошные линии, следует вызвать метод `setLineDash`, передав ему пустой массив.

Не принимающий параметров метод `getLineDash` возвращает заданный ранее массив с параметрами стиля линий.

Рисование дуг

Для рисования дуг (частей окружностей) используются два метода:

- `arc` — рисует дугу с центром в точке с координатами [*x*, *y*], указанными *радиусом*, *начальным* и *конечным углами*:

  ```
  arc(<x>, <y>, <радиус>, <начальный угол>, <конечный угол>[,
      <против часовой стрелки?>])
  ```

  Оба *угла* задаются в радианах и отсчитываются от горизонтальной оси. Если последним параметром передать значение `true`, дуга будет рисоваться против часовой стрелки, если задать `false` или вообще не указывать последний параметр — по часовой стрелке.

  Примеры:

  ```
  ctx1.beginPath();
  ctx1.arc(50, 50, 45, 0, Math.PI / 2);
  ctx1.stroke();
  ```

  ```
  ctx2.beginPath();
  ctx2.arc(50, 50, 45, 0, Math.PI / 2, true);
  ctx2.stroke();
  ```

  ```
  ctx3.beginPath();
  ctx3.arc(50, 50, 45, 0, Math.PI * 2);
  ctx3.stroke();
  ```

- `arcTo` — рисует дугу с указанным *радиусом* между двумя касательными. Первая касательная проводится между точкой, в которой находится перо, и точкой с координатами [*x1*, *y1*], вторая — между точками с координатами [*x1*, *y1*] и [*x2*, *y2*] (рис. 14.7).

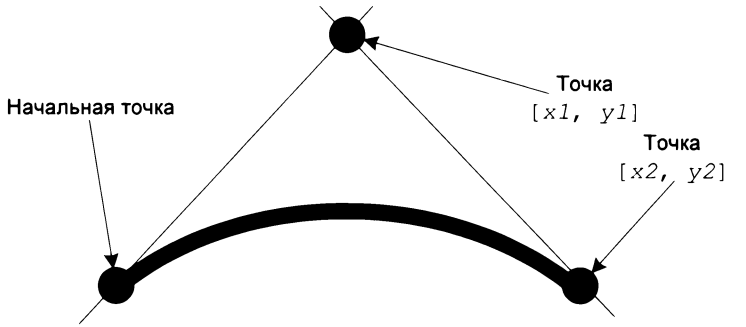


Рис. 14.7. Рисование дуги с указанным радиусом между двумя касательными

Формат:

```
arcTo(<x1>, <y1>, <x2>, <y2>, <радиус>)
```

Пример:

```
ctx4.beginPath();
ctx4.moveTo(5, 95);
ctx4.lineTo(5, 50);
ctx4.arcTo(5, 5, 50, 5, 45);
ctx4.arcTo(95, 5, 95, 50, 45);
ctx4.lineTo(95, 95);
ctx4.stroke();
```

Рисование кривых Безье

> *Кривая Безье* — кривая, описываемая начальной, конечной и одной (*квадратическая кривая Безье*) или двумя ключевыми (*кубическая кривая Безье*) точками.

Для рисования кривых Безье используются два метода:

- `quadraticCurveTo` — рисует квадратичную кривую Безье от точки, в которой находится перо, до точки [*x*, *y*]. Ключевая точка находится по координатам [*cx*, *cy*] (рис. 14.8).

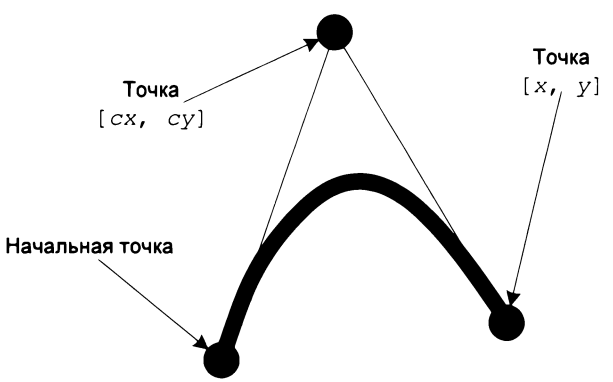


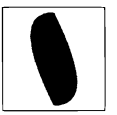
Рис. 14.8. Рисование квадратичной кривой Безье

Формат:

```
quadraticCurveTo(<cx>, <cy>, <x>, <y>)
```

Пример:

```
ctx1.beginPath();
ctx1.moveTo(50, 5)
ctx1.quadraticCurveTo(5, 5, 50, 95);
ctx1.quadraticCurveTo(95, 95, 50, 5);
ctx1.fill();
```



- `bezierCurveTo` — рисует кубическую кривую Безье от точки, в которой находится перо, до точки [*x*, *y*]. Ключевые точки находятся по координатам [*cx1*, *cy1*] и [*cx2*, *cy2*] (рис. 14.9).

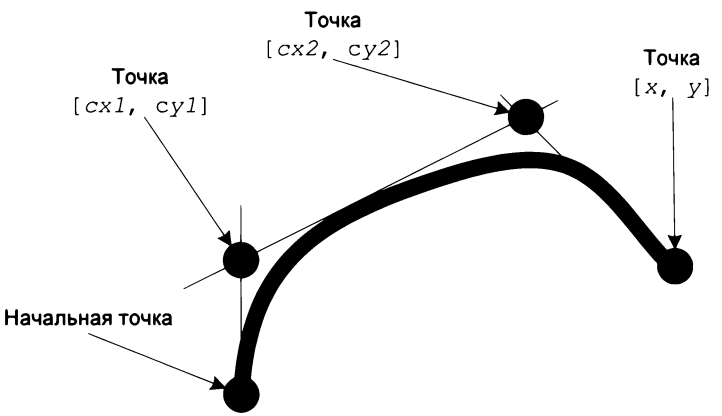


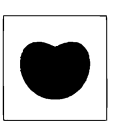
Рис. 14.9. Рисование кубической кривой Безье

Формат:

```
bezierCurveTo(<cx1>, <cy1>, <cx2>, <cy2>, <x>, <y>)
```

Пример:

```
ctx2.beginPath();
ctx2.moveTo(50, 30)
ctx2.bezierCurveTo(5, 5, 5, 80, 50, 80);
ctx2.moveTo(50, 30)
ctx2.bezierCurveTo(95, 5, 95, 80, 50, 80);
ctx2.fill();
```



Рисование прямоугольников

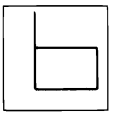
Чтобы нарисовать в составе сложной фигуры прямоугольник, нужно вызвать метод `rect`:

```
rect(<x>, <y>, <ширина>, <высота>)
```

Левый верхний угол прямоугольника будет находиться в точке [*x*, *y*]. По окончании рисования перо останется в той же точке.

Пример:

```
ctx1.beginPath();
ctx1.rect(30, 40, 60, 40);
ctx1.lineTo(30, 5);
ctx1.stroke();
```



14.7. Градиенты и графические закраски

Для рисования линий и заливок, помимо сплошных цветов, можно применять градиенты и графические закраски.

14.7.1. Линейные градиенты

Градиент — вид закраски, в котором один цвет постепенно переходит в другой, потом в третий и т. д.

Линейный градиент — градиент, в котором переходящие друг в друга цвета располагаются на прямой линии (рис. 14.10).

Ключевая точка — точка градиента, в которой присутствует «чистый» цвет.

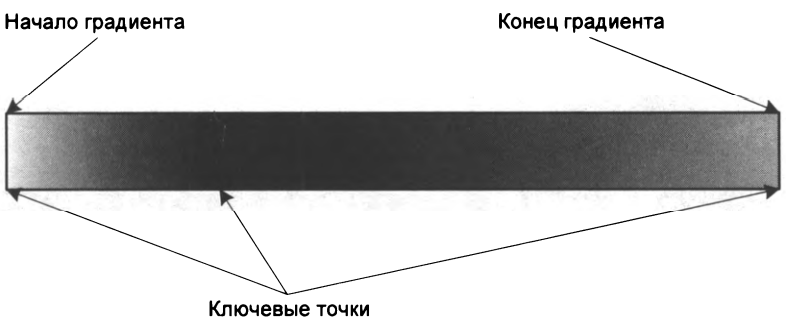


Рис. 4.10. Линейный градиент

Для создания самого линейного градиента применяется метод `createLinearGradient` графического контекста:

```
createLinearGradient(<x1>, <y1>, <x2>, <y2>)
```

Начало градиента будет находиться в точке [*x1*, *y1*], конец — в точке [*x2*, *y2*]. Эти координаты указываются относительно холста.

Метод возвращает объект класса `CanvasGradient`, представляющий созданный градиент.

Для добавления к градиенту ключевой точки у полученного объекта градиента нужно вызвать метод `addColorStop`:

```
addColorStop(<относительное расстояние>, <цвет>)
```

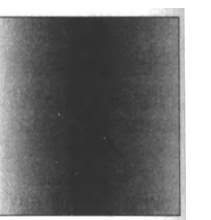
Относительное расстояние указывается в виде числа с плавающей точкой от `0.0` (начало градиента) до `1.0` (его конец), *цвет* — в любом формате, поддерживаемом CSS.

Полученный градиент присваивается свойствам графического контекста:

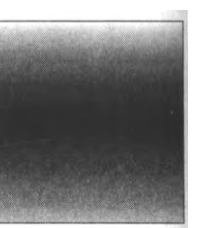
- `strokeStyle` — для закраски контуров;
- `fillStyle` — для закраски заливок.

Примеры:

```
const grd1 = ctx1.createLinearGradient(0, 0, 100, 0);
grd1.addColorStop(0, 'white');
grd1.addColorStop(0.5, 'black');
grd1.addColorStop(1, 'lightgrey');
ctx1.beginPath();
ctx1.rect(0, 0, 100, 100);
ctx1.fillStyle = grd1;
ctx1.fill();
```



```
const grd2 = ctx2.createLinearGradient(0, 0, 0, 100);
grd2.addColorStop(0, 'white');
grd2.addColorStop(0.5, 'black');
grd2.addColorStop(1, 'lightgrey');
ctx2.beginPath();
ctx2.rect(0, 0, 100, 100);
ctx2.fillStyle = grd2;
ctx2.fill();
```

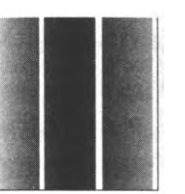


```
const grd3 = ctx3.createLinearGradient(0, 0, 100, 100);
grd3.addColorStop(0, 'white');
grd3.addColorStop(0.5, 'black');
grd3.addColorStop(1, 'lightgrey');
ctx3.beginPath();
ctx3.rect(0, 0, 100, 100);
ctx3.fillStyle = grd3;
ctx3.fill();
```

Сразу после своего создания градиент фиксируется на холсте, а отдельные части градиента проявляются на рисуемых с его применением фигурах. Это также относится к радиальному градиенту и графической закраске, рассматриваемым далее.

Пример:

```
const grd4 = ctx4.createLinearGradient(0, 0, 100, 0);
grd4.addColorStop(0, 'white');
grd4.addColorStop(0.5, 'black');
grd4.addColorStop(1, 'lightgrey');
ctx4.beginPath();
```



```
ctx4.rect(0, 0, 30, 100);
ctx4.rect(33, 0, 30, 100);
ctx4.rect(67, 0, 30, 100);
ctx4.fillStyle = grd4;
ctx4.fill();
```

14.7.2. Радиальные градиенты

Радиальный градиент — градиент, в котором переходящие друг в друга цвета располагаются на концентрических окружностях (рис. 14.11). Начало такого градиента находится на *начальной окружности*, конец — на *конечной*.

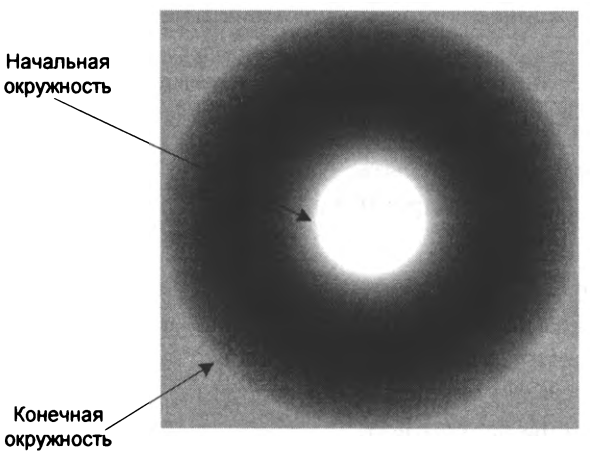


Рис. 14.11. Радиальный градиент

Для создания радиального градиента применяется метод `createRadialGradient` графического контекста:

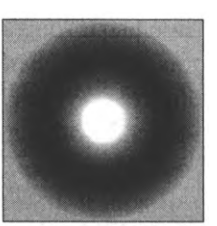
```
createRadialGradient(<x1>, <y1>, <R1>, <x2>, <y2>, <R2>)
```

[*x1*, *y1*] — координаты центра начальной окружности градиента, *R1* — ее радиус. [*x2*, *y2*] и *R2* — координаты центра и радиус его конечной окружности.

Возвращаемый результат — объект класса `CanvasGradient`, представляющий созданный градиент. Добавить ключевые точки можно вызовами метода `addColorStop`.

Примеры:

```
const grd5 = ctx5.createRadialGradient(50, 50, 10, 50, 50, 50);
grd5.addColorStop(0, 'white');
grd5.addColorStop(0.5, 'black');
grd5.addColorStop(1, 'lightgrey');
ctx5.beginPath();
ctx5.rect(0, 0, 100, 100);
ctx5.fillStyle = grd5;
ctx5.fill();
```



```
const grd6 = ctx6.createRadialGradient(0, 0, 20, 0, 0, 200);
grd6.addColorStop(0, 'white');
grd6.addColorStop(0.5, 'black');
grd6.addColorStop(1, 'lightgrey');
ctx6.beginPath();
ctx6.rect(0, 0, 100, 100);
ctx6.fillStyle = grd6;
```

14.7.3. Графическая закраска

> *Графическая закраска* — заполняет линии контура или заливки графическим изображением, повторяющимся или неповторяющимся.

Графическую закраску создает метод `createPattern` графического контекста:

```
createPattern(<изображение>, <режим повторения>)
```

В качестве *изображения* могут быть указаны собственно изображение (тег `<img>`), видеоролик (тег `<video>`) или другой холст. *Режим повторения* задается в виде одной из строк:

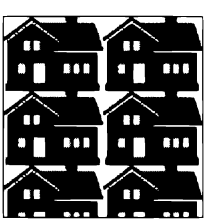
- `repeat` — повторять и по горизонтали, и по вертикали;
- `repeat-x` — повторять только по горизонтали;
- `repeat-y` — повторять только по вертикали;
- `no-repeat` — вообще не повторять.

> ***Примечание***
>
> Веб-обозреватель Mozilla Firefox при указании как значения `repeat-x`, так и значения `repeat-y` повторяет графическую заливку и по горизонтали, и по вертикали (как если бы было указано значение `repeat`).

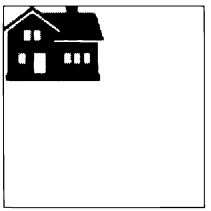
Метод `createPattern` вернет в качестве результата объект класса `CanvasPattern`, представляющий созданную графическую закраску. Ее следует присвоить свойству `strokeStyle` или `fillStyle`.

Примеры:

```
<img src="img.png" id="img">
. . .
const img = document.getElementById('img');
const ptn7 = ctx7.createPattern(img, 'repeat');
ctx7.beginPath();
ctx7.rect(0, 0, 100, 100);
ctx7.fillStyle = ptn7;
ctx7.fill();
```



```
const ptn8 = ctx8.createPattern(img, 'no-repeat');
ctx8.beginPath();
ctx8.rect(0, 0, 100, 100);
ctx8.fillStyle = ptn8;
ctx8.fill();
```



Создавать графическую закраску можно лишь после того, как закончится загрузка файла с изображением, используемым в качестве закраски. Добиться этого можно, поместив создающий закраску код в обработчик события `load` окна (или тега `<img>` с нужным изображением):

```
<img src="img.png" id="img">
. . .
const img = document.getElementById('img');
window.addEventListener('load', () => {
    const ptn7 = ctx7.createPattern(img, 'repeat');
    ctx7.beginPath();
    ctx7.rect(0, 0, 100, 100);
    ctx7.fillStyle = ptn7;
    ctx7.fill();
});
```

14.8. Преобразования

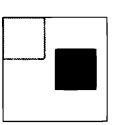
> *Преобразование* — манипуляция над системой координат холста: смещение ее начала, изменение масштаба или поворот.

Преобразования задают следующие методы графического контекста:

- `translate(<dx>, <dy>)` — смещает начало системы координат на величину *dx* по горизонтали и на величину *dy* по вертикали. Положительные величины вызывают смещение вправо и вниз, отрицательные — влево и вверх.

  Пример:

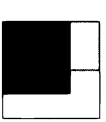
  ```
  ctx1.strokeRect(0, 0, 40, 40);
  ctx1.translate(50, 30)
  ctx1.fillRect(0, 0, 40, 40);
  ```

- `scale(<sh>, <sv>)` — изменяет масштаб системы координат до значения *sh* по горизонтали и до значения *sv* по вертикали. Величины масштаба задаются в виде числа с плавающей точкой: значения меньше `1` уменьшают масштаб, а больше `1` — увеличивают.

  Пример:

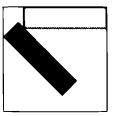
  ```
  ctx2.strokeRect(0, 0, 100, 50);
  ctx2.scale(0.7, 1.5)
  ctx2.fillRect(0, 0, 100, 50);
  ```

- `rotate(<угол>)` — поворачивает систему координат на указанный *угол*. *Угол* задается в радианах, положительные значения вызывают поворот по часовой стрелке, отрицательные — против.

  Пример:

```
ctx3.strokeRect(20, 0, 80, 20);
ctx3.rotate(Math.PI / 4)
ctx3.fillRect(20, 0, 80, 20);
```



14.9. Вывод сторонних изображений

Для вывода сторонних изображений применяется метод `drawImage` графического контекста. Он может быть вызван в трех форматах:

- `drawImage(<изображение>, <x>, <y>)` — выводит заданное *изображение*, располагая его левый верхний угол в точке с координатами [*x*, *y*], без изменения его размеров. В качестве *изображения* могут быть указаны собственно изображение (тег `<img>`), видеоролик (тег `<video>`) или другой холст.

  Пример:

```
<img src="img.png" id="img">
. . .
const img = document.getElementById('img');
ctx1.drawImage(img, 10, 20);
```

- `drawImage(<изображение>, <x>, <y>, <w>, <h>)` — то же самое, только выведенное *изображение* будет иметь ширину *w* и высоту *h*.

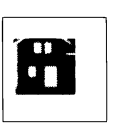
  Пример:

```
ctx2.drawImage(img, 10, 20, 90, 80);
```

- `drawImage(<изображение>, <fx>, <fy>, <fw>, <fh>, <x>, <y>, <w>, <h>)` — вырезает из *изображения* фрагмент с левым верхним углом в точке с координатами *[fx, fy]*, шириной *fw* и высотой *fh*, после чего выводит его в точке [*x*, *y*] холста, задав ему ширину *w* и высоту *h*.

  Пример:

```
ctx3.drawImage(img, 0, 10, 30, 27, 10, 20, 60, 54);
```



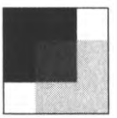
14.10. Управление композицией

Для управления композицией применяются два следующих свойства графического контекста:

- `globalAlpha` — задает уровень прозрачности рисуемой графики числом с плавающей точкой от `0.0` (полная прозрачность) до `1.0` (полная непрозрачность, значение по умолчанию).

  Пример:

```
ctx1.fillStyle = 'black';
ctx1.fillRect(0, 0, 70, 70);
ctx1.fillStyle = 'lightgrey';
ctx1.globalAlpha = 0.5;
ctx1.fillRect(30, 30, 70, 70);
```



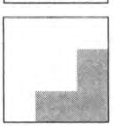
- `globalCompositeOperation` — указывает, как рисуемые фигуры накладываются на уже нарисованные. Значение указывается в виде одной из строк:

  - `source-over` — новая фигура рисуется поверх старой, выводятся обе фигуры (по умолчанию);

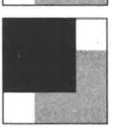
  - `source-atop` — выводятся лишь старая фигура и та часть новой, что накладывается на старую;

  - `source-in` — выводится лишь та часть новой фигуры, что накладывается на старую;

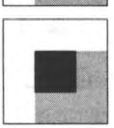
  - `source-out` — выводится лишь та часть новой фигуры, что *не* накладывается на старую;



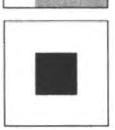
  - `destination-over` — новая фигура рисуется под старой, выводятся обе фигуры;



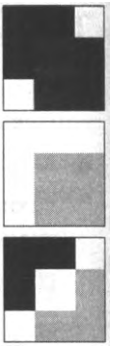
  - `destination-atop` — выводятся лишь новая фигура и та часть старой, что накладывается на новую;



  - `destination-in` — выводится лишь та часть старой фигуры, что накладывается на новую;



  - `destination-out` — выводится лишь та часть старой фигуры, что *не* накладывается на новую;



- `lighter` — то же, что и `source-over`, только часть новой фигуры, накладывающейся на старую, закрашивается цветом, полученным в результате сложения цветов обеих фигур;
- `copy` — выводится только новая фигура, старая удаляется;
- `xor` — выводятся только части новой и старой фигур, не накладывающиеся друг на друга.

Пример:

```
ctx12.fillStyle = 'black';
ctx12.fillRect(0, 0, 70, 70);
ctx12.fillStyle = 'lightgrey';
ctx12.globalCompositeOperation = 'xor';
ctx12.fillRect(30, 30, 70, 70);
```

14.11. Создание масок

> *Маска* — область, делающая видимой лишь часть холста. За пределами маски любая рисуемая графика остается невидимой.

Для создания маски нужно нарисовать контур сложной фигуры (как это сделать, описывалось в *разд. 14.6*), только в конце вместо метода `stroke` вызвать не принимающий параметров метод `clip`.

Пример:

```
ctx1.beginPath();
ctx1.rect(20, 0, 60, 100);
ctx1.clip();
ctx1.beginPath();
ctx1.arc(50, 50, 50, 0, 2 * Math.PI);
ctx1.fill();
```

14.12. Сохранение и восстановление состояния холста

> *Состояние холста* включает в себя параметры рисуемой графики (цвета линий и заливок, толщину линий, форму их концов и др.), преобразования системы координат и все созданные на холсте маски.

Метод `save` сохраняет состояние холста, а метод `restore` восстанавливает. Оба метода не принимают параметров:

```
// Сохраняем текущее состояние холста
ctx.save()
```

```
// Задаем параметры для рисования графики
// Рисуем графику
// Восстанавливаем состояние холста
ctx.restore();
// И тем самым возвращаемся к изначальным параметрам
```

Состояние холста можно сохранить неоднократно — последовательными вызовами метода `save`. Тогда при первом вызове метода `restore` будет восстановлено последнее сохраненное состояние холста, при втором — предпоследнее и т. д.

14.13. Упражнение. Рисуем цветок

Нарисуем, применяя полученные ранее навыки, вот такой цветок (рис. 14.12).

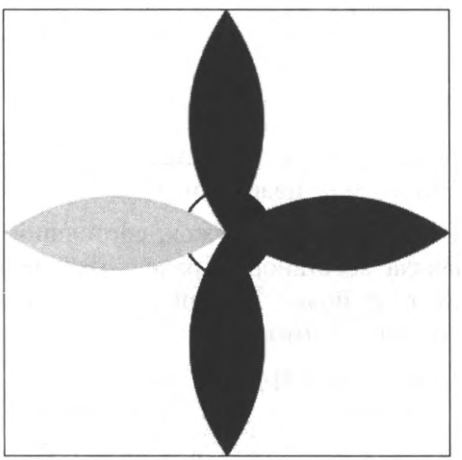


Рис. 14.12. Задание к упражнению из *разд. 14.13*

1. В папке 14\!sources сопровождающего книгу электронного архива (см. *приложение 3*) найдем файл 14.13.html и скопируем его куда-либо на локальный диск. В файле содержится заготовка для рисования цветка, включающая холст с якорем `draw` и размерами 300×300 пикселов.

2. Откроем копию файла 14.13.html в текстовом редакторе и добавим в конец HTML-кода тег `<script>` с первыми выражениями сценария:

```
<script type="text/javascript">
    const colors = ['red', 'green', 'blue', 'yellow'];
    const draw = document.getElementById('draw');
    const ctx = draw.getContext('2d');
</script>
```

   Они объявят массив с наименованиями цветов, которыми будут закрашены отдельные лепестки, получат доступ к холсту, а потом к его графическому контексту.

3. Добавим выражение, смещающее центр системы координат в центр холста, — так удобнее рисовать:

```
ctx.translate(150, 150);
```

4. Напишем выражение, сохраняющее состояние холста:

```
ctx.save();
```

Впоследствии мы сможем вернуться к этому состоянию, просто восстановив его.

5. Напишем код, рисующий лепестки:

```
for (let i = 0; i < 4; ++i) {
    ctx.beginPath();
    ctx.fillStyle = colors[i];
    ctx.moveTo(0, 0);
    ctx.quadraticCurveTo(-50, -75, 0, -150);
    ctx.quadraticCurveTo(50, -75, 0, 0);
    ctx.fill();
    ctx.rotate(Math.PI / 2);
}
```

Каждый лепесток закрашивается очередным цветом из массива `colors`, после чего система координат поворачивается на 90°.

6. Осталось нарисовать центральный кружок, состоящий из двух половинок: черной и белой. Но сначала восстановим ранее сохраненное состояние и укажем композицию, при которой новые фигуры рисуются под старыми (поскольку кружок находится под лепестками).

Теперь напишем код, рисующий кружок:

```
ctx.restore();
ctx.globalCompositeOperation = 'destination-over';
ctx.beginPath();
ctx.arc(0, 0, 30, -Math.PI / 2, Math.PI / 2);
ctx.fill();
ctx.beginPath();
ctx.arc(0, 0, 29, Math.PI / 2, 1.5 * Math.PI);
ctx.stroke();
```

Откроем страницу 14.13.html в веб-обозревателе и полюбуемся на нарисованный цветок.

14.14. Самостоятельные упражнения

- Создайте у диаграммы из *разд. 14.5* (см. рис. 14.5) вертикальную шкалу, проградуированную в процентах, с делениями каждые 10%.
- Закрасьте столбцы той же диаграммы линейным градиентом, в котором цвета изменяются от светло-серого до черного в направлении снизу вверх.

У вас должно получиться так (рис. 14.13).

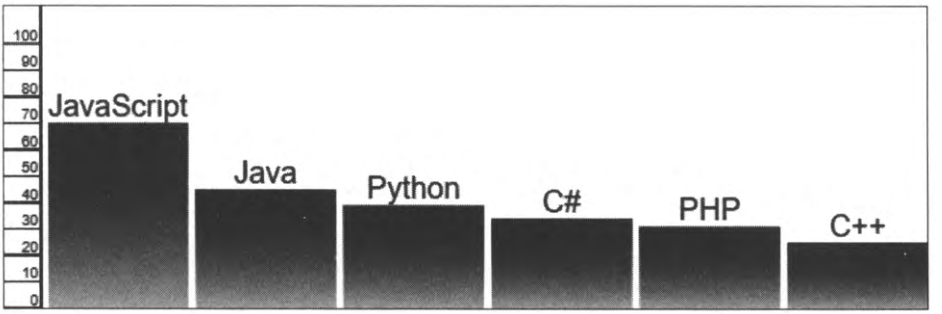


Рис. 14.13. Столбчатая диаграмма из *разд. 14.5*, дополненная вертикальной шкалой, проградуированной в процентах, и закрашенная линейным градиентом

Урок 15
Объявление своих классов

Конструкторы
Создание свойств и методов
Прототипы классов
Динамические свойства
Наследование
Расширение встроенных классов

15.1. Объявление нового класса

В ряде случаев удобно реализовать какую-либо часть функциональности — например, сложный элемент страницы, — в виде своего собственного класса. Так мы сведем все инструменты для управления этой частью функциональности воедино.

15.1.1. Конструктор. Объявление свойств и методов в конструкторе

Для объявления нового класса достаточно написать для него функцию-конструктор.

> *Конструктор* — функция, которая будет выполняться при создании каждого нового объекта представляемого ей класса.

Конструктор может принимать произвольное количество параметров, но не должен возвращать результат. Имя конструктора, представляющего вновь объявленный класс, станет именем этого класса.

Конструктор объявляет у создаваемого объекта необходимые свойства и присваивает им изначальные значения: фиксированные, рассчитанные в результате вычислений или полученные с параметрами. Ссылка на создаваемый объект хранится в переменной `this`, доступной в теле конструктора.

> В качестве конструктора может быть задействована только *именованная* или *анонимная* функция. Функцию-стрелку использовать нельзя, т. к. в ее теле переменная `this` ссылается не на текущий объект, а на объект текущего окна.

Свойства у создаваемого объекта объявляются простым присваиванием им значений (как и добавленные свойства, описанные в *разд. 4.12*).

Чтобы объявить метод, следует создать у объекта свойство и присвоить ему функцию — именованную или анонимную (но, опять же, не функцию-стрелку!), которая и реализует этот метод. В этой функции также будет доступна переменная `this`, хранящая ссылку на текущий объект.

В качестве примера объявим класс `User` со свойствами `name` (имя пользователя), `password` (пароль), `age` (возраст), `sendNotification` (если `true`, пользователь хочет получать оповещения об обновлениях на сайте, если `false` — не хочет) и методом `isMature`, возвращающим `true`, если возраст пользователя не меньше 18 лет.

```
// Функция User__isMature(), реализующая метод isMature класса User
function User__isMature() {
    return this.age >= 18;
}
// Функция-конструктор класса User
// В качестве параметров принимает имя, пароль и возраст пользователя,
// заносит их в свойства создаваемого объекта и объявляет метод
function User(name, password, age) {
    this.name = name;
    this.password = password;
    this.age = age;
    this.sendNotification = true;
    this.isMature = User__isMature;
}
```

Объект объявленного таким образом класса может быть создан с применением оператора `new`:

```
new <имя класса>([<параметры конструктора через запятую>])
```

Не забываем, что имя класса всегда совпадает с именем конструктора.

Вот пример создания и использования объектов класса `User`:

```
const user1 = new User('ivan_ivanov', '123456789', 45);
const user2 = new User('petr_petrov', '987654321', 17);
let user1Age = user1.age;
user2.password = 'sUp3rPa$$WoЯd';
if (user1.isMature()) {
    // Если user1 взрослый, пускаем его на сайт
}
```

15.1.2. Прототип. Объявление свойств и методов в прототипе

Свойства и методы, которые следует объявить в классе, также можно занести в его прототип.

> *Прототип* — служебный объект класса `Object`, чьи свойства и методы наследуются объектом класса, которому принадлежит этот прототип. Наследование свойств и методов из прототипа выполняется еще до исполнения конструктора.

Каждый из классов, как встроенных в JavaScript, DOM, BOM или HTML API, так и объявленных программистом, содержит свой прототип. Прототип любого объявленного программистом класса изначально пуст. Ссылка на прототип хранится в статическом свойстве `prototype` владеющего им класса.

В прототипе имеет смысл объявлять свойства, чьи значения не будут изменяться в конструкторе (в противном случае проще объявить их непосредственно в конструкторе).

Вот пример объявления того же класса `User`, в котором свойство `sendNotifications` и метод `isMature` объявлены в прототипе:

```
function User2(name, password, age) {
    this.name = name;
    this.password = password;
    this.age = age;
}
// Объявляем в прототипе свойство sendNotifications и метод isMature
User2.prototype.sendNotifications = true;
User2.prototype.isMature = function() {
    return this.age >= 18;
};
```

Где объявлять свойства и методы класса: в конструкторе или прототипе — дело вкуса. Первый способ несколько нагляднее (поскольку все объявления собраны в одном месте), но второй дает некоторую прибавку в производительности (т. к. свойства и методы объявляются в прототипе лишь единожды и потом просто копируются в создаваемые объекты).

15.2. Упражнение. Объявляем класс слайдера

В *разд. 6.5* мы написали слайдер. Перепишем его, реализовав в виде класса `Slider`. Ради упрощения сделаем новый слайдер без вывода миниатюр.

1. В папке 15\!sources сопровождающего книгу электронного архива (см. *приложение 3*) найдем файл 15.2.html и папку images и скопируем их куда-либо на локальный диск. Страница 15.2.html содержит заготовку для создания слайдера. В папке images хранятся графические файлы с 1.jpg по 7.jpg.
2. Откроем копию страницы 15.2.html в текстовом редакторе и найдем HTML-код, создающий слайдер:

```
<section>
    <img src="images/1.jpg">
</section>
<nav id="slider1">
    <a href="images/1.jpg" class="active">1</a>
    <a href="images/2.jpg">2</a>
    <a href="images/3.jpg">3</a>
</nav>
```

Условимся о следующем:

- набор «кнопок» слайдера будем идентифицировать по указанному у него якорю (вспомним, что название «кнопки» у слайдера взято в кавычки, поскольку это не настоящие кнопки, создаваемые тегом `<input>`). Как можно видеть, у нашего набора «кнопок» — тега `<nav>` — задан якорь `slider1`;
- в наборе «кнопок» должны присутствовать только гиперссылки, создающие «кнопки»;
- перед набором «кнопок» должен присутствовать элемент, содержащий в качестве первого потомка тег `<img>` (в нем будет выводиться очередное изображение).

Следование этим соглашением позволит нам заметно упростить код слайдера.

3. Допишем в конце HTML-кода тег `<script>`, в который поместим объявление конструктора класса `Slider`:

```
<script type="text/javascript">
    function Slider(buttonset) {
        this.buttonset = buttonset;
        this.buttonset.addEventListener('click', Slider__showImage);
    }
</script>
```

Конструктор в качестве параметра примет ссылку на объект набора «кнопок». В теле конструктора сохраняем этот элемент в свойстве `buttonset` и привязываем к нему обработчик события `click` — функцию `Slider__showImage`, которую сейчас напишем.

4. Перед конструктором вставим объявление функции `Slider__showImage`, которая станет выводить при щелчках на «кнопках» указанные в них изображения:

```
function Slider__showImage(evt) {
    if (evt.target != this) {
        this.previousElementSibling.firstElementChild.src =
            evt.target.href;
        let el;
        for (let i = 0; i < this.childElementCount; ++i) {
            el = this.children[i];
            el.className = (el == evt.target) ? 'active' : '';
        }
        evt.preventDefault();
    }
}
```

Если щелчок был выполнен на одной из «кнопок» набора (а не на самом наборе), выводим относящееся к «кнопке» изображение в теге `<img>` — первом потомке предыдущего соседа набора «кнопок». После чего перебираем все «кнопки», делаем активной ту, на которой был выполнен щелчок, а остальные деактивируем. И не забываем отменить обработку события по умолчанию.

5. В целом написанный нами слайдер полностью функционален. Но что-то он слишком прост... Добавим классу `Slider` два метода:
   - `getCurrentIndex` — не принимает параметров и возвращает номер активной «кнопки»;
   - `setCurrentIndex` — принимает в качестве единственного параметра номер «кнопки», выводит соответствующее изображение и делает «кнопку» активной.
6. Допишем код, объявляющий метод `getCurrentIndex`:

```
Slider.prototype.getCurrentIndex = function () {
    for (let i = 0;
        i < this.buttonset.childElementCount; ++i) {
        if (this.buttonset.children[i].classList.contains('active'))
            return i;
    }
};
```

   Здесь перебираются все «кнопки» в наборе и возвращается номер первой из них, к которой привязан стилевой класс `active`.
7. Добавим объявление метода `setCurrentIndex`:

```
Slider.prototype.setCurrentIndex = function (ind) {
    if (ind < 0)
        ind = 0;
    if (ind > this.buttonset.childElementCount - 1)
        ind = this.buttonset.childElementCount - 1;
    this.buttonset.previousElementSibling.firstElementChild.src =
        this.buttonset.children[ind].href;
    for (let i = 0; i < this.buttonset.childElementCount; ++i)
        this.buttonset.children[i].className =
            (i == ind) ? 'active' : '';
};
```

   Если номер меньше `0`, полагаем его равным `0`, а если больше номера последней из «кнопок» — равным номеру последней «кнопки». Далее извлекаем ссылку на графический файл из «кнопки» с указанным номером и выводим этот файл в теге `<img>`. Наконец делаем эту «кнопку» активной, привязав к ней стилевой класс `active`.
8. Добавим выражение, создающее объект класса `Slider` на основе набора «кнопок» с якорем `slider1`:

```
const slider1 = new Slider(document.getElementById('slider1'));
```

9. Откроем страницу 15.2.html в веб-обозревателе, пощелкаем на «кнопках» и убедимся, что слайдер работает. То есть, реализовав слайдер в виде класса, мы сможем управлять им программно, просто обращаясь к его свойствам и вызывая его методы. Это важное преимущество объектного подхода.

10. Для пробы добавим выражение, которое сразу после открытия страницы выведет в слайдере второе по счету изображение, вызвав у объекта слайдера метод `setCurrentIndex`:

```
slider1.setCurrentIndex(1);
```

Обновим страницу и убедимся, что метод выполнился успешно.

11. Еще одно преимущество объектного подхода: возможность без проблем создать на странице сразу нескольких слайдеров.

Проверим, так ли это, для чего вставим сразу после HTML-кода первого слайдера фрагмент, создающий второй слайдер:

```
<p></p>
<section>
    <img src="images/7.jpg">
</section>
<nav id="slider2">
    <a href="images/4.jpg">1</a>
    <a href="images/5.jpg">2</a>
    <a href="images/6.jpg">3</a>
    <a href="images/7.jpg"
class="active">4</a>
</nav>
```

Разделим слайдеры пустым абзацем. А во втором слайдере сделаем изначально активной последнюю «кнопку».

12. В код сценария добавим выражение, создающее объект второго слайдера:

```
const slider2 = new Slider(document.
            getElementById('slider2'));
```

Обновим страницу и проверим, работают ли слайдеры. Если мы не допустили ошибок, оба слайдера должны функционировать независимо друг от друга (рис. 15.1).

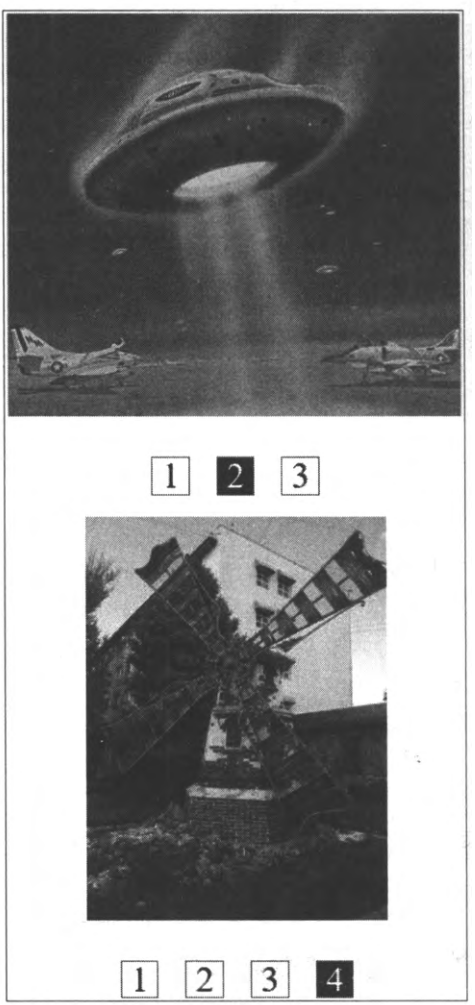


Рис. 15.1. На веб-странице размещены два независимых слайдера

15.3. Упражнение. Создаем динамическое свойство

В классе `Slider`, написанном в *разд. 15.2*, мы объявили два метода: `getCurrentIndex`, возвращающий номер активной «кнопки», и `setCurrentIndex`, активизирующий «кнопку» с указанным номером. Перепишем этот класс, заменив эти методы динамическим свойством `currentIndex`. Это упростит использование класса.

> *Динамическое свойство* — имитация обычного свойства класса, представляющее собой комбинацию из методов геттер и сеттер.

> Метод *геттер* — вызывается при попытке получить значение динамического свойства. Не принимает параметров и возвращает результат, который и становится значением свойства. Возвращаемое значение может как извлекаться из другого, обычного, свойства, так и получаться в результате каких-либо вычислений.

> Метод *сеттер* — вызывается при попытке присваивания значения свойству. Принимает один параметр — новое значение свойства — и не возвращает результата. Полученное значение может присваиваться другому, обычному, свойству или использоваться в каких-либо вычислениях.

Динамическое свойство объявляется вызовом статического метода `defineProperty` класса `Object` в формате:

`defineProperty(`*<объект>*`, `*<имя свойства>*`, `*<объект конфигурации>*`).`

В качестве *объекта* следует указать прототип нужного класса. *Имя свойства* задается в виде строки. *Объект конфигурации*, относящийся к классу `Object`, должен иметь свойства `get` и `set`, задающие, соответственно, геттер и сеттер.

В нашем коде уже есть готовые геттер и сеттер — методы `getCurrentIndex` и `setCurrentIndex`. Нам остается превратить их в обычные функции и указать в вызове метода `defineProperty`.

1. В коде сценария страницы 15.2.html найдем объявления методов `getCurrentIndex` и `setCurrentIndex` и переделаем их в объявления функций `Slider__getCurrentIndex` и `Slider__setCurrentIndex` (исправленный код выделен полужирным шрифтом):

```
function Slider__getCurrentIndex() {
    . . .
}
function Slider__setCurrentIndex(ind) {
    . . .
}
```

2. Запишем под объявлениями этих функций код, объявляющий динамическое свойство `currentIndex`:

```
Object.defineProperty(Slider.prototype, 'currentIndex', {
    get: Slider__getCurrentIndex,
    set: Slider__setCurrentIndex
});
```

3. Исправим выражение, выводящее в первом слайдере второе по счету изображение:

```
slider1.currentIndex = 1;
```

При присваивании динамическому свойству `currentIndex` нового значения будет выполнена функция `Slider__setCurrentIndex`, которая переключит слайдер на второе изображение. И наоборот, при попытке извлечь значение из этого свойства выполнится функция `Slider__getCurrentIndex`, вычисляющая и возвращающая номер активной «кнопки».

Откроем страницу 15.2.html в веб-обозревателе и убедимся, что оба слайдера работают, а первый слайдер изначально выводит второе изображение.

Пользоваться динамическим свойством удобнее, чем аналогичной парой методов, поскольку запомнить имя одного свойства проще.

15.3.1. Динамические свойства, доступные только для чтения и только для записи

В *объекте конфигурации*, передаваемом методу `defineProperty` в третьем параметре, необязательно указывать и геттер, и сеттер:

- если указать только геттер — свойство будет доступно только для чтения:

```
Object.defineProperty(SomeClass.prototype, 'readOnlyProperty',
    {get: SomeClass__getROP});
. . .
let val = someClassObject.readOnlyProperty;
someClassObject.readOnlyProperty = 5;            // Ошибка!
```

- если указать только сеттер — свойство станет доступным только для записи:

```
Object.defineProperty(SomeClass.prototype, 'writeOnlyProperty',
    {set: SomeClass__setWOP});
someClassObject.writeOnlyProperty = 5;
let val = someClassObject.writeOnlyProperty;     // Ошибка!
```

15.3.2. Объявление обычных свойств методом *defineProperty*

Статический метод `defineProperty` класса `Object` можно применять для объявления обычных свойств, указав имя создаваемого свойства во втором параметре. При этом можно задать ряд дополнительных параметров, создав в *объекте конфигурации* следующие свойства:

- `value` — изначальное значение свойства;
- `writable` — если `true`, свойство будет доступно для чтения и записи (по умолчанию), если `false` — только для чтения;
- `enumerable` — если `true`, свойство будет доступно для перебора в цикле по свойствам, описанном в *разд. 4.13* (по умолчанию), если `false` — не будет доступно;
- `configurable` — если `true`, параметры свойства могут быть изменены последующим вызовом метода `defineProperty` (по умолчанию), если `false` — не могут.

Вот пример объявления свойства `selector` класса `Slider`, со значением `'.slider'`, доступного только для чтения и с неизменяемыми параметрами:

```
Object.defineProperty(Slider.prototype, 'selector', {
    value: '.slider',
    writable: false,
    configurable: false
});
```

15.4. Упражнение. Реализуем наследование классов

Напишем класс `Slider2` — более развитую разновидность слайдера. Его конструктор сможет принимать в качестве параметра, помимо объекта набора «кнопок», еще и якорь. А сам класс получит поддержку методов `goNext` и `goPrevious`, выводящих в слайдере, соответственно, следующее и предыдущее изображения. Чтобы не писать заново весь код, унаследуем класс `Slider2` от класса `Slider`.

> *Наследование* — создание одного класса на основе другого. Унаследованный класс (*производный*, или *подкласс*), помимо собственных свойств и методов, получает все свойства и методы класса, от которого он наследован (*базового*, или *суперкласса*).

1. В папке 15\!sources сопровождающего книгу электронного архива (см. *приложение 3*) найдем файл 15.4.html и папку images и скопируем их куда-либо на локальный диск. Страница 15.4.html содержит код класса `Slider` и заготовку для создания слайдера.
2. Откроем страницу 15.4.html в текстовом редакторе и добавим к коду сценария объявление функции-конструктора класса `Slider2`:

```
function Slider2(buttonset) {
    if (typeof buttonset == 'string')
        buttonset = document.getElementById(buttonset);
}
```

Если в качестве параметра получена строка, подразумевается, что был передан якорь набора «кнопок», по которому следует найти сам набор.

3. Чтобы производный класс унаследовал все свойства и методы базового класса, следует указать в качестве прототипа производного класса копию прототипа базового класса.

   > Создать копию любого объекта (не только прототипа) можно вызовом статического метода `create` класса `Object` в формате:
   >
   > `create(<`*`копируемый объект`*`>)`.
   >
   > Метод возвращает полную копию копируемого объекта.

   Итак, добавим выражение, задающее в качестве прототипа класса `Slider2` копию прототипа класса `Slider`:

   ```
   Slider2.prototype = Object.create(Slider.prototype);
   ```

4. В полученной таким образом копии прототипа базового класса следует указать ссылку на конструктор производного класса. Ссылка на конструктор хранится в свойстве `constructor` прототипа.

   Добавим выражение, задающее правильный конструктор в прототипе производного класса:

   ```
   Slider2.prototype.constructor = Slider2;
   ```

5. В конструкторе производного класса необходимо вызвать конструктор базового класса, который выполнит инициализацию слайдера и подготовку его к работе. Конструктор базового класса должен быть вызван в контексте объекта производного класса, чтобы переменная `this` в теле вызываемого конструктора указывала на этот объект.

   > Вызвать какую-либо функцию в контексте указанного объекта с передачей ей заданных параметров можно с помощью метода `apply`, вызываемого непосредственно у этой функции в формате:
   >
   > `apply(<`*`объект, в контексте которого вызывается функция`*`>[,`
   > *`<массив с параметрами>`*`]).`

   В теле вызываемой этим методом функции переменная `this` будет ссылаться на указанный в вызове *`объект`*. Вызванной функции будут переданы все параметры из заданного *`массива`* (часто в его качестве указывают объект-коллекцию `arguments`, описанную в *разд. 4.8*).

   Добавим в тело конструктора класса `Slider2` выражение, вызывающее конструктор базового класса `Slider` (выделено полужирным шрифтом):

   ```
   function Slider2(buttonset) {
       . . .
       Slider.apply(this, arguments);
   }
   ```

6. Добавим объявления методов `goNext` и `goPrevious` нового класса:

   ```
   Slider2.prototype.goNext = function () {
       this.currentIndex = this.currentIndex + 1;
   };
   ```

```
Slider2.prototype.goPrevious = function () {
    this.currentIndex = this.currentIndex - 1;
};
```

7. Напишем выражение, создающее объект слайдера:

```
const slider1 = new Slider2('slider1');
```

8. Откроем страницу 15.4.html в веб-обозревателе и проверим слайдер в работе.

   Затем проверим в работе методы `goNext` и `goPrevious`, объявленные в новом классе.

9. Вставим под HTML-кодом слайдера абзац с кнопками **Назад** и **Вперед**:

```
<p><input type="button" id="previous" value="Назад">
<input type="button" id="next" value="Вперед"></p>
```

10. Допишем в код сценария фрагмент, обрабатывающий нажатия на этих кнопках:

```
const previous = document.getElementById('previous');
const next = document.getElementById('next');
previous.addEventListener('click', () => {
    slider1.goPrevious();
});
next.addEventListener('click', () => {
    slider1.goNext();
});
```

Обновим страницу, пощелкаем на кнопках **Назад** и **Вперед** и посмотрим, что произойдет со слайдером.

15.4.1. Переопределение методов

> *Переопределение метода* — дополнение функциональности метода базового класса, выполненное в производном классе.

Выполняется переопределение метода в три этапа:

- в производном классе объявляется метод с тем же именем, что и у переопределяемого метода базового класса;
- в теле объявленного метода записывается код, реализующий дополнительную функциональность;
- в нужном месте кода метода выполняется вызов одноименного метода базового класса, для чего применяется описанный ранее метод `apply`.

Вот пример переопределения метода `someMethod` базового класса `BaseClass` в производном классе `DerivedClass`:

```
// Объявление базового класса
function BaseClass(par1, par2) {
    . . .
}
```

```
// Объявление метода
BaseClass.prototype.someMethod = function (arg1, arg2) {
    . . .
};

// Объявление производного класса
function DerivedClass(par1, par2, par3) {
    . . .
    BaseClass.apply(this, [par1, par2]);
}
// Объявление переопределяемого метода
DerivedClass.prototype.someMethod = function (arg1, arg2) {
    . . .
    BaseClass.prototype.someMethod.apply(this, arguments);
};
```

15.4.2. Использование метода *call* для вызова методов базового класса

Для вызова метода базового класса при их переопределении можно применять и метод `call`:

```
call(<объект, в контексте которого вызывается функция>[,
    <параметры через запятую>])
```

Единственное его отличие от метода `apply` — параметры, передаваемые вызываемому методу, записываются непосредственно в вызове метода `call`.

Пример:

```
function DerivedClass(par1, par2, par3) {
    . . .
    BaseClass.call(this, par1, par2);
}
DerivedClass.prototype.someMethod = function (arg1, arg2) {
    . . .
    BaseClass.prototype.someMethod.call(this, arg1, arg2);
};
```

15.5. Объявление статических свойств и методов

Статическое свойство объявляется в виде добавленного свойства у самой функции-конструктора.

Пример объявления статического свойства `selector` класса `Slider`:

```
function Slider(buttonset) {
    . . .
}
Slider.selector = '.slider';
```

Объявление доступного только для чтения статического свойства `selector` класса `Slider`:

```
Object.defineProperty(Slider, 'selector',
    {value: '.slider', writable: false});
```

> *Статический метод* объявляется в виде метода самой функции-конструктора.

Пример объявления статического метода `createSliders` класса `Slider`:

```
Slider.createSliders = function (selector) {
    . . .
}
```

Объявленные таким образом статические свойства и методы не наследуются подклассами, поскольку не являются частью прототипа суперкласса.

15.6. Упражнение. Расширяем функциональность встроенных классов

Класс коллекции `HTMLCollection`, описанный в *разд. 7.1*, не поддерживает исключительно полезный метод `forEach` (см. *разд. 4.5*). Давайте добавим его в класс коллекции.

> Новые свойства и методы объявляются в прототипе нужного класса. Таким образом можно расширить функциональность любого класса, встроенного в JavaScript, DOM, BOM или в HTML API.

1. Откроем страницу 15.4.html в текстовом редакторе.
2. В самом начале кода сценария, перед объявлением функции `Slider__showImage`, вставим код, объявляющий в классе `HTMLCollection` метод `forEach`:

```
HTMLCollection.prototype.forEach = function (func, ths) {
    for (let i = 0; i < this.length; ++i)
        if (ths)
            func.call(ths, this[i], i, this);
        else
            func(this[i], i, this);
};
```

3. Исправим код функции `Slider__showImage`, использовав в нем только что объявленный метод (исправленный код выделен полужирным шрифтом):

```
function Slider__showImage(evt) {
    if (evt.target != this) {
        this.previousElementSibling.firstElementChild.src =
            evt.target.href;
        this.children.forEach((el) => {
            el.className = (el == evt.target) ? 'active' : '';
        });
        evt.preventDefault();
    }
}
```

Как видим, благодаря применению метода `forEach` для перебора коллекции, код несколько сократился.

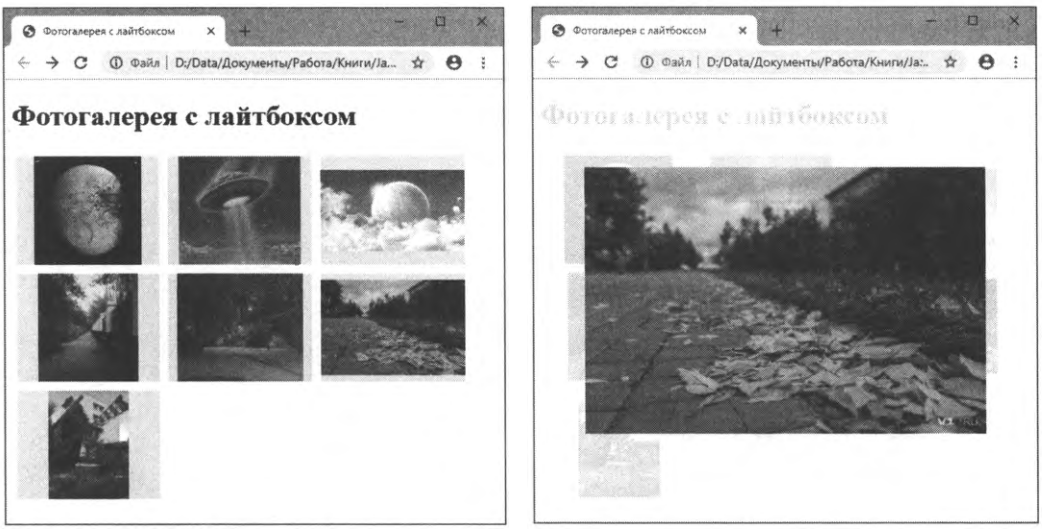
Откроем страницу 15.4.html в веб-обозревателе и проверим работу слайдера.

15.7. Самостоятельные упражнения

- В сценарии на странице 15.4.html объявите класс `Slider3`, производный от класса `Slider2`. Переопределите его методы:
  - `goNext` — чтобы при достижении последнего изображения он выводил первое изображение;
  - `goPrevious` — чтобы при достижении первого изображения он выводил последнее.
- Там же в классе `Slider3` объявите:
  - статическое свойство `selector` — хранящее строку с CSS-селектором, с которым должны совпадать наборы «кнопок» (изначальное значение `'nav.slider'`);
  - статический метод `createSliders` — не принимающий параметров, преобразующий в слайдеры все наборы «кнопок», что совпадают с заданным в свойстве `selector` CSS-селектором, и возвращающий массив созданных слайдеров.
- Найдите в папке 15\!sources сопровождающего книгу электронного архива (см. *приложение 3*) страницу 15.7.html с фотогалереей, в которой выведены миниатюры изображений. При щелчке на любой миниатюре откроется лайтбокс (рис. 15.2) с полной редакцией изображения.

> *Лайтбокс* — панель, обычно полупрозрачная, выводящаяся поверх страницы, занимающая все внутреннее пространство окна веб-обозревателя, показывающая какой-либо фрагмент содержимого и исчезающая при щелчке мышью или по нажатию клавиши <Esc>. Часто применяется для вывода полной редакции изображения при щелчке на его миниатюре.

Изучите программный код фотогалереи и реализуйте ее в виде класса `PhotoGallery`. Объявите в этом классе доступное только для чтения динамическое свойство `lightboxOpened`, хранящее `true`, если лайтбокс открыт, и `false` — в противном случае.



а *б*

Рис. 15.2. *а* — лайтбокс скрыт; *б* — лайтбокс открыт

Урок 16
Компоненты

Компоненты и их написание
Замыкания
Изоляция компонентов

16.1. Понятие компонента

Обычный класс слайдера, написанный в *уроке 15*, требует наличия на странице готового, созданного вручную набора элементов: тега `<section>` с тегом `<img>` внутри, панели навигации `<nav>` с набором гиперссылок. Если мы допустим ошибку в HTML-коде, слайдер не заработает.

В случае компонента нам не придется создавать все эти элементы вручную.

> *Компонент* — класс, реализующий сложный элемент страницы и конструирующий все необходимые элементы самостоятельно.

Конструктор класса-компонента принимает два параметра:

- ссылку на элемент страницы, обычно пустой, в котором будут размещены сконструированные элементы (*базовый элемент*);
- массив или объект с исходными данными для конструирования компонента.

Конструирование всех необходимых элементов, задание их параметров и привязка обработчиков событий — одним словом, *инициализация* компонента, — выполняется в его конструкторе.

Применение компонентов вместо обычных классов упрощает программирование и устраняет множество ошибок, связанных с неправильным набором сложного HTML-кода. Нужно лишь создать пустой базовый элемент, подготовить исходные данные и создать объект компонента.

Компоненты часто оформляются в виде библиотек, рассчитанных на применение сторонними разработчиками в своих проектах. В этом случае компонент настоятельно рекомендуется сделать изолированным (чем мы займемся чуть позже).

16.2. Упражнение. Создаем компонент

Преобразуем слайдер, написанный в *разд. 15.3*, в компонент, который:

- при создании получит (помимо ссылки на базовый элемент) исходные данные — служебный объект класса `Object` со свойствами `images` (массив ссылок на графи-

ческие файлы) и `initial` (номер изначально активной «кнопки». Если это свойство не указано, будет активна первая «кнопка»);

- привяжет к базовому элементу стилевой класс `slider`, после чего к нему будут применены оформляющие CSS-стили;
- создаст в базовом элементе «кнопки», а над ними — панель для вывода изображений.

* * *

1. Найдем в папке 16\!sources сопровождающего книгу электронного архива (см. *приложение 3*) файл 16.2.html и папку images и скопируем их в какую-либо папку локального диска. Страница 16.2.html содержит полностью работающий слайдер, основанный на коде из *разд. 15.3*. В папке images находятся графические файлы.
2. Откроем копию страницы 16.2.html в текстовом редакторе и исправим селекторы стилей, оформляющие слайдер (здесь и далее добавленный и исправленный код выделен полужирным шрифтом, а удаленный — зачеркиванием):

```
~~section, nav~~.slider, section.slider-panel {
    . . .
}
section.slider-panel {
    . . .
}
section.slider-panel img {
    . . .
}
~~nav~~.slider {
    . . .
}
~~nav~~.slider a {
    . . .
}
~~nav~~.slider a.active {
    . . .
}
```

Сам набор «кнопок» у нас будет помечаться стилевым классом `slider`. Панель для вывода изображений мы сделаем из тега `<section>` со стилевым классом `slider-panel`.

3. Найдем фрагмент HTML-кода, создающий слайдер, удалим имеющийся там тег `<section>` и содержимое тега `<nav>` (все это будет создаваться программно):

```
~~<section>~~
~~    <img src="images/1.jpg">~~
~~</section>~~
<nav id="slider1">
```

```
    <a href="images/1.jpg" class="active">1</a>
    <a href="images/2.jpg">2</a>
    <a href="images/3.jpg">3</a>
</nav>
```

У нас должен остаться лишь пустой тег `<nav>`, который станет базовым элементом для слайдера.

Теперь займемся контроллером класса `Slider`. Переделки там будут значительными, и мы рассмотрим их по частям.

4. Добавим в список параметров конструктора параметр `config`, которому будет присвоен объект с исходными данными:

```
function Slider(buttonset, config) {
    . . .
}
```

5. Добавим в конец кода конструктора выражение, задающее номер активной «кнопки» по умолчанию (`0`, т. е. первая «кнопка»), если он явно не указан в свойстве `initial` объекта с исходными данными:

```
function Slider(buttonset, config) {
    . . .
    if (!config.initial)
        config.initial = 0;
}
```

6. Добавим выражение, привязывающее к базовому элементу стилевой класс `slider`:

```
function Slider(buttonset, config) {
    . . .
    if (!config.initial)
        config.initial = 0;
    this.buttonset.classList.add('slider');
}
```

7. Добавим код, который создаст «кнопки» слайдера:

```
function Slider(buttonset, config) {
    . . .
    let button;
    config.images.forEach((el, i) => {
        button = document.createElement('a');
        button.href = el;
        button.textContent = i + 1;
        if (i == config.initial)
            button.classList.add('active');
        this.buttonset.appendChild(button);
    });
}
```

Здесь перебираются все элементы массива из свойства `images` объекта с исходными данными, и для каждого элемента создается «кнопка». Активная «кнопка» выделяется стилевым классом `active`.

8. Добавим код, создающий панель для вывода изображений, вкладывающий в нее тег `<img>`, в котором, собственно, и будут выводиться изображения, и помещающий панель над базовым элементом:

```
function Slider(buttonset, config) {
    . . .
    const panel = document.createElement('section');
    panel.classList.add('slider-panel');
    const img = document.createElement('img');
    img.src = config.images[config.initial];
    panel.appendChild(img);
    this.buttonset.insertAdjacentElement('beforebegin', panel);
}
```

В теге `<img>` изначально выводим изображение, соответствующее активной «кнопке».

На этом написание компонента слайдера закончено. Осталось исправить код, создающий его объект.

9. Найдем код, создающий объект слайдера, и переделаем его так:

```
const cnfg1 = {
    images: ['images/4.jpg', 'images/5.jpg', 'images/6.jpg',
        'images/7.jpg']
};
const slider1 = new Slider(document.getElementById('slider1'), cnfg1);
```

Мы создали объект `cnfg1`, хранящий исходные данные (массив ссылок на файлы с изображениями), и передали его конструктору класса `Slider`.

Откроем страницу 16.2.html в веб-обозревателе и посмотрим на наш слайдер (рис. 16.1).

10. Добавим в объявление объекта `cnfg1` свойство `initial` со значением `2`, чтобы слайдер сразу же переключился на третью «кнопку»:

```
const cnfg1 = {
    images: ['images/4.jpg', 'images/5.jpg', 'images/6.jpg',
        'images/7.jpg'],
    initial: 2
};
```

Обновим страницу и проверим, действительно ли изначально выводится третье по счету изображение (рис. 16.2).

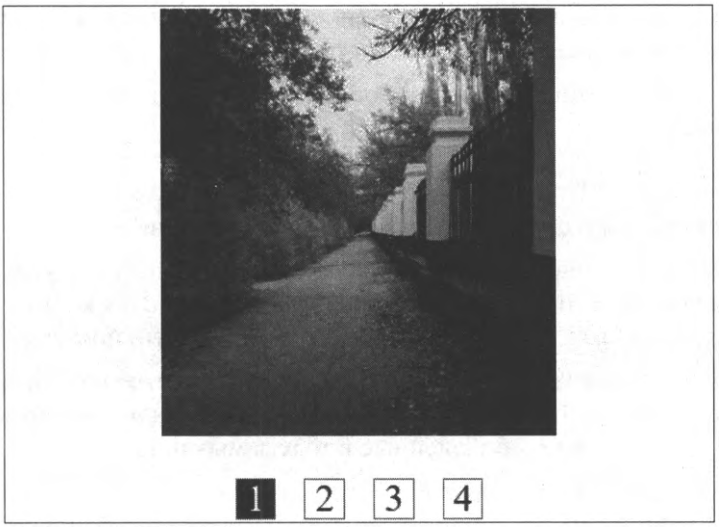


Рис. 16.1. Слайдер, преобразованный в компонент

Рис. 16.2. Слайдер-компонент сразу после создания переключился на 3-ю кнопку

16.3. Замыкания

В составе кода нашего компонента-слайдера есть обычные функции: `Slider__showImage`, `Slider__getCurrentIndex` и `Slider__setCurrentIndex`. Сторонний разработчик, применяющий библиотеку с нашим классом, по незнанию может вызвать одну из них, что приведет к программной ошибке и даже нарушит работу слайдера.

Разработчик может задействовать и другую библиотеку, в которой также объявлены функции: `Slider__showImage`, `Slider__getCurrentIndex` и `Slider__setCurrentIndex`.

Тогда функции, объявленные в одной библиотеке, переопределят их «тезок» из другой библиотеки, и последняя перестанет работать.

Поэтому код нашего компонента следует изолировать от остального кода, заключив в замыкание.

> *Замыкание* — анонимная функция, которая:
>
> - объявляется и сразу же вызывается на исполнение;
> - содержит в своем теле объявления переменных и функций, которые должны быть изолированы от остального кода (поскольку они являются ее локальными переменными и вложенными функциями);
> - обязательно возвращает одну функцию в качестве результата. Тогда возвращенная функция удержит в памяти все локальные переменные и вложенные функции, объявленные в теле замыкания.

Замыкание записывается в формате:

```
(function () {
 <тело замыкания>
})()
```

Пустые круглые скобки `()`, поставленные в самом конце, — это оператор вызова функции, запускающий ее на выполнение. Объявление функции, включая слово `function`, заключено в круглые скобки, чтобы веб-обозреватель сначала выполнил его, а уже потом исполнил оператор вызова.

> Функцию, возвращенную замыканием, необходимо присвоить какой-либо переменной.

Классический пример простого замыкания, содержащего переменную, возвращающего функцию и реализующего счетчик:

```
const counter = (function () {
    // Эта переменная будет изолирована от "внешнего" кода
    let count = 0;
    return function () {
        return ++count;
    };
})();
counter();                          // 1
counter();                          // 2
counter();                          // 3
// Пытаемся обратиться к переменной count, объявленной в замыкании
// и хранящей текущее значение счетчика
let cnt = count;                    // Ошибка!
// Пытаемся присвоить переменной count новое значение
count = 10000;
// В результате будет создана глобальная переменная count, а значение
// переменной count из замыкания останется неизменным
counter();                          // 4
```

Если требуется изолировать в замыкании класс, *возвращаемым результатом должна быть его функция-конструктор.*

16.4. Упражнение. Изолируем компонент в замыкании

Сделаем компонент слайдера `Slider`, написанный в *разд. 16.2*, полностью изолированным от остального кода.

1. Откроем страницу 16.2.html и заключим код компонента `Slider` (объявления конструктора `Slider`, функций `Slider__showImage`, `Slider__getCurrentIndex`, `Slider__setCurrentIndex` и вызов метода `defineProperty`) в замыкание (добавленный и исправленный код здесь и далее выделен полужирным шрифтом):

```
(function () {
    function Slider__showImage(evt) {
        . . .
    }
    function Slider(buttonset, config) {
        . . .
    }
    function Slider__getCurrentIndex() {
        . . .
    }
    function Slider__setCurrentIndex(ind) {
        . . .
    }
    Object.defineProperty( . . . );
})();
```

2. Добавим в тело замыкания выражение, возвращающее в качестве результата функцию-конструктор:

```
(function () {
    . . .
    function Slider(buttonset, config) {
        . . .
    }
    . . .
    Object.defineProperty( . . . );
    return Slider;
})();
```

3. Допишем перед замыканием фрагмент, который присвоит возвращенный конструктор глобальной константе `Slider`:

```
const Slider = (function () {
    . . .
})();
```

Мы дали этой константе имя, совпадающее с именем конструктора. Так нам не придется переписывать код, создающий объекты этого компонента.

Откроем страницу 16.2.html в веб-обозревателе и посмотрим, как чувствует себя компонент слайдера, теперь полностью автономный.

16.5. Самостоятельные упражнения

- В сценарии страницы 16.2.html объявите изолированный компонент `Slider2` — производный от класса компонента `Slider`. Добавьте в класс `Slider2` поддержку методов `goNext` и `goPrevious`.
- Превратите фотогалерею с лайтбоксом, сделанную на странице 15.7.html, в компонент. В качестве исходных данных пусть он принимает служебный объект класса `Object` со свойством `images`, в котором хранится массив ссылок на графические файлы.
- Вынесите код компонентов `Slider` и `Slider2` в файл сценария slider.js, а таблицу стилей, задающую оформление, — в файл slider.css, сделав тем самым эти компоненты готовыми к распространению.
- Сделайте то же самое с компонентом `PhotoGallery`, вынеся его код в файлы photogallery.js и photogallery.css.

ЧАСТЬ IV

Взаимодействие с сервером

- Технология AJAX.
- Серверные веб-приложения.
- Платформа PHP.
- Фронтенды и бэкенды.
- Веб-службы.
- Серверные сообщения.

Урок 17
Технология AJAX

Реализация AJAX
Формат JSON

17.1. Реализация AJAX

Существует возможность программно загрузить с веб-сервера произвольный файл с целью вывести его содержимое на веб-странице или использовать в вычислениях. Эту возможность обеспечивает технология AJAX.

> *AJAX* (Asynchronous JavaScript и XML, асинхронные JavaScript и XML) — технология, позволяющая загружать с веб-сервера файлы любого типа. Инструменты для ее реализации входят в состав HTML API.

Загружаемые посредством AJAX файлы могут хранить фрагменты HTML-кода (их можно непосредственно вывести на веб-странице), данные в формате JSON (этот формат будет рассмотрен далее), XML-данные, обычный текст и др. Применение AJAX позволяет быстрее обновлять страницы, не загружая их повторно.

> ***Пояснение***
>
> XML (eXtensible Markup Language, расширяемый язык разметки) — язык для кодирования произвольных данных. Похож на HTML — XML-данные представляются в виде набора вложенных друг в друга парных тегов и их атрибутов.
>
> Ранее активно применялся для кодирования пересылаемых по Сети данных, но в середине 2000-х годов был вытеснен более компактным и простым в обработке форматом JSON. В этой книге XML не рассматривается.

> В целях безопасности посредством AJAX разрешается загружать файлы *только с веб-сервера*. С локального диска загрузить файл таким образом невозможно.

Для тестирования написанных сценариев мы будем использовать пакет хостинга XAMPP, содержащий в своем состава веб-сервер Apache HTTP Server. Установка этого пакета описывается в *приложении 2*.

Реализация AJAX включает несколько последовательно выполняемых этапов, описываемых далее.

17.1.1. Подготовка объекта AJAX

Программная загрузка по технологии AJAX выполняется объектом класса `XMLHttpRequest`. Этот объект следует создать явно — оператором `new` без указания каких-либо параметров:

```
const ajax = new XMLHttpRequest();
```

17.1.2. Указание интернет-адреса загружаемого файла

Интернет-адрес загружаемого файла (и пара других параметров) задается в вызове метода `open` созданного ранее объекта AJAX:

```
open(<метод пересылки данных>, <интернет-адрес>, <асинхронная загрузка?>)
```

Наименование *метода пересылки данных* указывается в виде строки `'GET'` или `'POST'`. Для простой загрузки файла следует указать метод GET.

Интернет-адрес загружаемого файла также задается в виде строки. Третьим параметром передается логическая величина:

- `false` — будет выполнена синхронная загрузка, при которой исполнение кода приостановится, пока запрошенный файл не будет загружен;
- `true` — будет выполнена асинхронная загрузка, при которой сразу после отправки веб-серверу запроса на загрузку исполнение кода продолжится.

Применение синхронной загрузки упрощает программирование, однако страница на момент выполнения загрузки перестает реагировать на действия пользователя. При использовании асинхронной загрузки страница не «подвисает», но программирование несколько усложняется.

Тем не менее, в последнее время чаще всего применяется асинхронная загрузка. Ее будем использовать и мы:

```
ajax.open('GET', '/fragments/fr3.html', true);
```

Синхронную же загрузку и ее реализацию мы рассмотрим позже.

Метод `open` просто задает параметры загрузки, но не запускает ее. Для запуска загрузки следует вызвать метод `send`, речь о котором пойдет далее.

17.1.3. Получение загруженного файла

При асинхронной загрузке момент получения запрошенного файла предугадать невозможно. Поэтому о его получении и обработке следует подумать заранее, еще до отправки запроса на загрузку.

Получение загруженного файла реализуется в обработчике события `readystatechange` объекта AJAX, выполняющего загрузку. Это событие возникает при каждом изменении состояния загрузки, поэтому при каждом его возникновении необходимо проверять значения двух свойств того же объекта:

- свойство `readyState` должно хранить значение `4`;
- свойство `status` должно хранить значение `200`.

Если оба этих условия выполняются, строковое содержимое загруженного файла можно извлечь из свойства `responseText` того же объекта AJAX:

```
ajax.addEventListener('readystatechange', function () {
    if (this.readyState == 4 && this.status == 200)
        // Выводим содержимое загруженного HTML-файла в элементе output
        output.innerHTML = this.responseText;
});
```

Свойство `readyState` хранит числовой код состояния объекта AJAX в виде одного из следующих числовых значений:

- `0` — запрос не инициализирован (еще не был вызван метод `open`);
- `1` — запрос инициализирован, но не отправлен (метод `open` был вызван, метод `send` — еще нет);
- `2` — получена начальная часть серверного ответа с запрошенным файлом;
- `3` — идет загрузка запрошенного файла;
- `4` — запрошенный файл полностью загружен.

Свойство `status` хранит числовой код состояния обработки запроса, отправленный веб-сервером. Практически всегда оно принимает следующие значения:

- `200` — запрос успешно обработан, и запрошенный файл выслан;
- `403` — веб-сервер отклонил запрос, поскольку доступ к запрошенному файлу запрещен;
- `404` — запрошенный файл не существует.

> ***Примечание***
>
> Полный список всех возможных кодов статуса можно найти на странице **http://www.restapitutorial.ru/httpstatuscodes.html**.

Свойство `statusText` объекта AJAX хранит строку с описанием состояния — например: `'OK'` или `'Not Found'`.

17.1.4. Отправка веб-серверу запроса на получение файла

Отправку запроса на загрузку файла производит метод `send` объекта AJAX, который в этом случае вызывается без параметров:

```
ajax.send();
```

17.2. Упражнение. Пишем сверхдинамический веб-сайт

Напишем сверхдинамический сайт, представляющий сведения о трех текстовых редакторах, применяемых программистами.

> *Сверхдинамический веб-сайт* — состоит из одной страницы, на которой выводятся различные фрагменты содержания, хранящиеся в отдельных HTML-файлах. Загрузка и вывод файлов с фрагментами выполняется программно при щелчках на гиперссылках.

Сверхдинамические сайты работают быстрее обычных, т. к. относительно небольшие файлы с HTML-фрагментами загружаются быстрее громоздких файлов с полноценными веб-страницами.

> ***Примечание***
>
> Для тестирования этого сайта понадобится установленный и работающий пакет хостинга XAMPP. Процесс его установки описан в *приложении 2*.

1. Найдем в папке 17\!sources сопровождающего книгу электронного архива (см. *приложение 3*) файл 17.2.html и папку fragments. Скопируем их в корневую папку веб-сервера Apache из состава пакета XAMPP, расположенную по пути: c:\xampp\htdocs.
2. Файл 17.2.html хранит заготовку для создания единственной страницы сайта, а папка fragments — файлы atom.html, notepadpp.html и vsc.html, в которых хранятся фрагменты содержимого.
3. Откроем любой файл из папки fragments — например, atom.html:

```
<h2>Atom</h2>
<p>Текстовый редактор для программистов.</p>
<p>Достоинства:</p>
. . .
<p><a href="https://atom.io/">Официальный сайт</a></p>
```

   Здесь нет ни шапки сайта, ни его поддона, ни панели навигации, ни таблицы стилей — только уникальное для этого фрагмента содержание. Поэтому файлы с фрагментами так компактны.

4. Откроем копию страницы 17.2.html и найдем код, создающий панель навигации с гиперссылками и элемент, в котором будут выводиться фрагменты содержания:

```
<nav>
    <a href="fragments/vsc.html">Visual Studio Code</a>
    <a href="fragments/atom.html">Atom</a>
    <a href="fragments/notepadpp.html">Notepad++</a>
</nav>
<section id="output"></section>
```

   Ссылки на файлы с фрагментами записаны непосредственно в тегах `<a>`. Элементу для вывода фрагментов дан якорь `output`.

5. Добавим в конец HTML-кода тег `<script>` с выражениями, получающими доступ к гиперссылкам и элементу `output`:

```
<script type="text/javascript">
    const links = document.querySelectorAll('nav a');
    const output = document.getElementById('output');
</script>
```

6. Добавим код, который создаст объект AJAX и привяжет к его событию `readystatechange` обработчик, выводящий содержимое загруженного фрагмента в элементе `output`:

```
const ajax = new XMLHttpRequest();
ajax.addEventListener('readystatechange', function () {
    if (this.readyState == 4 && this.status == 200)
        output.innerHTML = this.responseText;
});
```

7. Напишем функцию `loadHTML`, которая будет обрабатывать щелчки мышью на гиперссылках, и привяжем ее в качестве обработчика события `click`:

```
function loadHTML(evt) {
    ajax.open('GET', this.href, true);
    ajax.send();
    evt.preventDefault();
}
links.forEach((el) => {
    el.addEventListener('click', loadHTML);
});
```

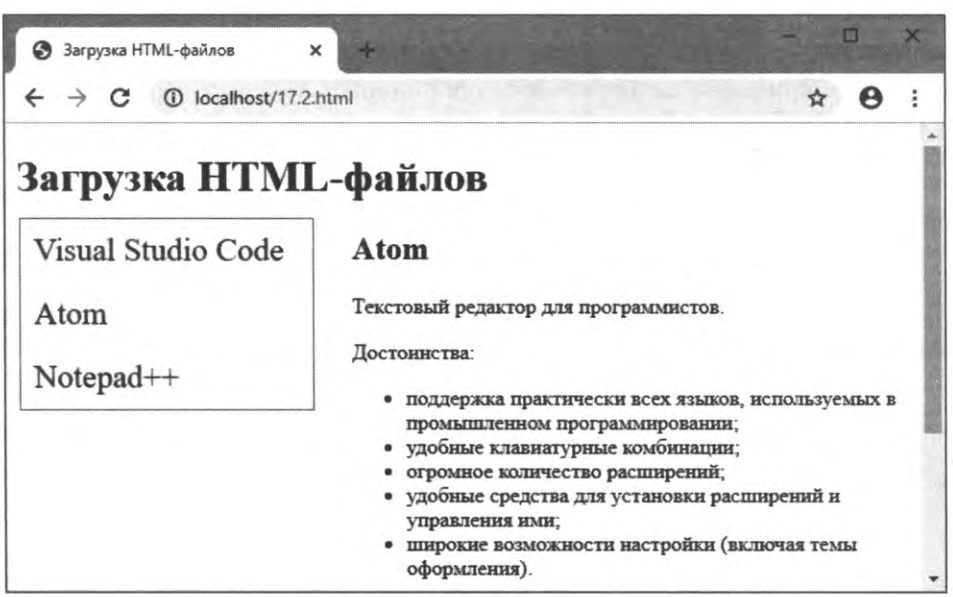


Рис. 17.1. Был выполнен щелчок на гиперссылке **Atom**

Ссылку на загружаемый файл мы извлекаем из атрибута `href` тега гиперссылки, на которой был выполнен щелчок мышью.

Откроем панель управления XAMPP и запустим веб-сервер. В веб-обозревателе перейдем по интернет-адресу **http://localhost/17.2.html** и, когда страница откроется, щелкнем на любой гиперссылке (рис. 17.1).

17.3. Синхронная загрузка файлов по технологии AJAX

Если методу `open` объекта AJAX передать третьим параметром значение `false`, веб-обозреватель выполнит синхронную загрузку. В этом случае исполнение сценария приостановится на вызове метода `send`, пока запрашиваемый файл не будет получен. Содержимое загруженного файла станет доступным сразу же после вызова метода `send`:

```
ajax.open('GET', '/fragments/fr3.html', false);
ajax.send();
// Ждем, пока файл не будет загружен
output.innerHTML = ajax.responseText;
```

Тем не менее, рассмотренный ранее подход — с обработкой события `readystatechanged` — будет работать и при синхронной загрузке.

Синхронную загрузку рекомендуется использовать, когда время загрузки файла гарантированно невелико, а посетители сайта готовы немного подождать — например, при программировании корпоративных решений.

17.4. Формат JSON

Формат JSON применяется для кодирования данных, которым перед выводом требуется программная обработка. Данные в этом формате компактны, легко кодируются и декодируются.

> *JSON* (JavaScript Object Notation, объектная нотация JavaScript) — текстовый формат обмена данными, основанный на объектной нотации JavaScript (см. *разд. 4.9*).

Данные в формате JSON очень похожи на JavaScript-код, объявляющий служебный объект класса `Object`. Различия между JSON и JavaScript состоят в следующем:

- строки заключаются только в двойные кавычки (одинарные не допускаются);
- имена свойств также заключаются в двойные кавычки;
- поддерживаются лишь следующие типы данных:
  - строки;
  - числа;
  - логические величины;

- массивы (заключаются в квадратные скобки);
- служебные объекты класса `Object` (заключаются в фигурные скобки);
- `null`.

Временны́е отметки в виде объектов класса `Date`, функции и значение `undefined` не поддерживаются.

JSON-данные хранятся в обычных текстовых файлах в кодировке UTF-8 с расширениями json.

Вот пример сведений об уровне популярности различных языков программирования, записанных в формате JSON:

```
{
    "title": "Уровень популярности различных языков программирования
            за указанный год",
    "year": 2018,
    "data": [
        {"name": "JavaScript", "level": 70},
        {"name": "Java", "level": 45},
        {"name": "Python", "level": 39}
    ]
}
```

Декодирование полученных от веб-сервера JSON-данных, которые можно извлечь из того же свойства `responseText`, выполняется вызовом статического метода `parse` класса `JSON`:

```
parse(<данные в формате JSON>)
```

В качестве результата возвращается служебный объект класса `Object`, содержащий декодированные, готовые к обработке данные:

```
let jsonData;
ajax.addEventListener('readystatechange', function () {
    if (this.readyState == 4 && this.status == 200)
        jsonData = JSON.parse(this.responseText);
});
```

17.5. Упражнение. Загрузка и вывод JSON-данных

Напишем страницу, которая будет загружать JSON-файл с данными о распространенности различных языков программирования и выводить их в табличном виде.

1. В папке 17\!sources сопровождающего книгу электронного архива (см. *приложение 3*) найдем файлы 17.5.html и 17.5.json и скопируем их в корневую папку веб-сервера из пакета XAMPP. Файл 17.5.html хранит страницу с необходимым начальным кодом, а файл 17.5.json — непосредственно JSON-данные.

2. Откроем копию файла 17.5.json в текстовом редакторе и посмотрим на структуру хранящихся в нем данных (она понадобится нам для написания обрабатывающего их кода):

```
{
    "year": 2018,
    "data": [
        {"name": "JavaScript", "level": 70},
        {"name": "Java", "level": 45},
        {"name": "Python", "level": 39},
        {"name": "C#", "level": 34},
        {"name": "PHP", "level": 31},
        {"name": "C++", "level": 25}
    ]
}
```

3. Откроем копию файла 17.5.html в текстовом редакторе и посмотрим на код, создающий «заготовку» таблицы:

```
<p>Год: <span id="year"></span></p>
<table>
    <thead>
        <tr>
            <th>Язык</th>
            <th colspan="2">Популярность, %</th>
        </tr>
    </thead>
    <tbody id="output">
    </tbody>
</table>
```

Во встроенном контейнере (теге `<span>`) с якорем `year` будет выведен год, к которому относятся данные. В секции основного содержания таблицы (теге `<tbody>`) с якорем `output` станут выводиться строки со сведениями о языках программирования. Каждая строка включит три ячейки:

- название языка;
- уровень его популярности, выраженный числом;
- встроенный контейнер (`<span>`), создающий столбец импровизированной диаграммы, ширина которого показывает уровень популярности языка.

4. Добавим в конце HTML-кода тег `<script>` с выражениями, получающими доступ к элементам `year`, `output` и создающими объект AJAX:

```
<script type="text/javascript">
    const year = document.getElementById('year');
    const output = document.getElementById('output');
    const ajax = new XMLHttpRequest();
</script>
```

5. Код, конструирующий таблицу со сведениями о языках программирования, весьма велик, и его удобнее вынести в отдельную функцию.

   Добавим объявление функции `showData`, выводящей эту таблицу:

```
function showData() {
    let tr, td, span;
    let languages = JSON.parse(ajax.responseText);
    year.textContent = languages.year;
    languages.data.forEach((el) => {
        tr = document.createElement('tr');
        td = document.createElement('td');
        td.textContent = el.name;
        tr.appendChild(td);
        td = document.createElement('td');
        td.textContent = el.level;
        tr.appendChild(td);
        td = document.createElement('td');
        span = document.createElement('span');
        span.style.width = el.level + '%';
        td.appendChild(span);
        tr.appendChild(td);
        output.appendChild(tr);
    });
}
```

В свойстве `level` каждого языка хранится уровень его популярности в процентах. Эту величину мы можем использовать для указания ширины столбца диаграммы.

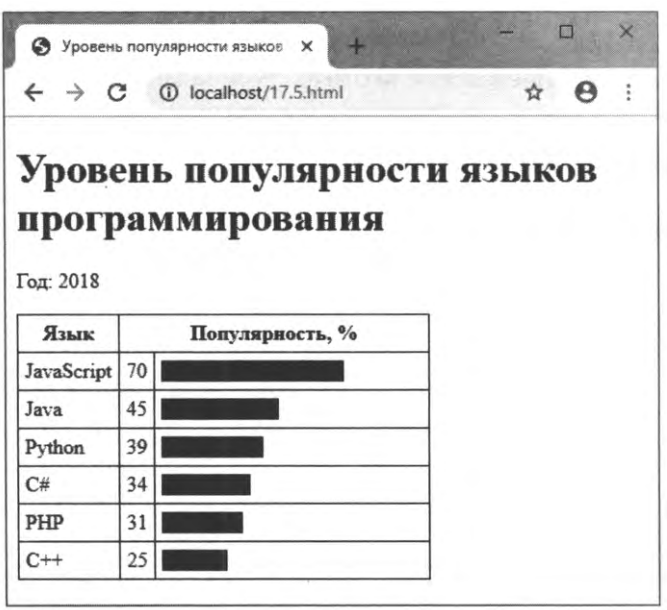


Рис. 17.2. Данные, загруженные из JSON-файла, выведены в табличном виде

6. Добавим код, который после загрузки файла 17.5.json вызовет функцию `showData`:

```
ajax.addEventListener('readystatechange', function () {
    if (this.readyState == 4 && this.status == 200)
        showData();
});
```

7. И напишем код, загружающий этот файл:

```
ajax.open('GET', '17.5.json', true);
ajax.send();
```

Запустим веб-сервер, откроем веб-обозреватель и выполним переход по интернет-адресу **http://localhost/17.5.html** (рис. 17.2).

17.6. Упражнение. Пишем компонент *AJAXLoader*

Чтобы упростить дальнейшую работу, реализуем код, выполняющий загрузку данных по AJAX, в виде невидимого компонента `AJAXLoader` (написание компонентов рассматривалось в *уроке 16*).

> *Невидимый компонент* — компонент, не выводящийся на экран.

Компонент `AJAXLoader` будет поддерживать следующие свойства и методы:

- `GET` — статическое свойство, доступное только для чтения, хранит обозначение метода GET;
- `POST` — статическое свойство, доступное только для чтения, хранит обозначение метода POST;
- `load(<`*интернет-адрес*`>, <`*функция*`>)` — метод, запускающий загрузку файла с указанного *интернет-адреса* и вызывающий заданную *функцию* после его успешной загрузки. *Функция* в качестве единственного параметра получит содержимое загруженного файла.

* * *

1. Создадим в корневой папке веб-сервера из пакета XAMPP файл ajaxloader.js и запишем в него код, объявляющий замыкание:

```
const AJAXLoader = (function () {
})();
```

2. Вставим в замыкание объявление функции-конструктора (добавленный код здесь и далее выделен полужирным шрифтом):

```
const AJAXLoader = (function () {
    function AJAXLoader() {
        this.ajax = new XMLHttpRequest();
        this.ajax.addEventListener('readystatechange', dataLoaded);
    }
})();
```

Конструктор создает объект AJAX, присваивает его свойству `ajax` и привязывает к нему в качестве обработчика события `readystatechange` функцию `dataLoaded`, которую мы объявим позже.

3. Добавим в замыкание объявление свойств `GET` и `POST`:

```
const AJAXLoader = (function () {
    . . .
    Object.defineProperty(AJAXLoader, 'GET',
        {value: 'GET', writable: false});
    Object.defineProperty(AJAXLoader, 'POST',
        {value: 'POST', writable: false});
})();
```

4. Добавим в замыкание объявление метода `load`:

```
const AJAXLoader = (function () {
    . . .
    AJAXLoader.prototype.load = function (url, success) {
        this.ajax.success = success;
        this.ajax.open(AJAXLoader.GET, url, true);
        this.ajax.send();
    };
})();
```

Функцию, обрабатывающую загруженный файл и получаемую методом со вторым параметром, мы присвоили добавленному свойству `success` объекта AJAX. Эта функция далее будет вызвана в обработчике события `readystatechange`.

5. Добавим в замыкание выражение, возвращающее конструктор в качестве результата:

```
const AJAXLoader = (function () {
    . . .
    return AJAXLoader;
})();
```

6. Вставим перед конструктором объявление функции `dataLoaded`:

```
const AJAXLoader = (function () {
    function dataLoaded() {
        if (this.readyState == 4)
            if (this.status == 200)
                this.success(this.responseText);
    }
    function AJAXLoader() {
        . . .
    }
    . . .
})();
```

В случае успешной загрузки файла вызывается функция, полученная ранее методом `load` и сохраненная в добавленном свойстве `success` объекта AJAX. Функции будет передано содержимое загруженного файла.

7. Переделаем страницу 17.5.html так, чтобы находящийся в ее коде сценарий использовал компонент `AJAXLoader`.

   Для этого откроем страницу 17.5.html в текстовом редакторе и добавим в секцию заголовка (тег `<head>`) тег `<script>`, привязывающий файл сценария ajaxloader.js:

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

8. Удалим весь код сценария, кроме выражений, получающих доступ к элементам `year` и `output`, и объявления функции `showData` (показан код, который должен остаться после удаления):

```
const year = document.getElementById('year');
const output = document.getElementById('output');
function showData() {
    . . .
}
```

9. Исправим объявление функции `showData` следующим образом:

```
function showData(response) {
    let tr, td, span;
    let languages = JSON.parse(response);
    . . .
}
```

10. Добавим в конец сценария код, создающий объект компонента `AJAXLoader` и запускающий загрузку файла:

```
const ajax = new AJAXLoader();
ajax.load('17.5.json', showData);
```

Запустим веб-сервер и перейдем в веб-обозревателе по интернет-адресу **http://localhost/17.5.html**. Проверим, правильно ли отображается страница.

17.7. Самостоятельные упражнения

- Реализуйте в компоненте `AJAXLoader` обработку ситуации, когда файл не был успешно загружен. Добавьте в метод `load` необязательный третий параметр, с которым будет передаваться функция, вызываемая в этом случае и получающая в качестве единственного параметра текст состояния (его можно получить из свойства `statusText` объекта AJAX). Если же в вызове метода третий параметр не был указан, пусть на экран выводится окно-сообщение с текстом состояния.

- Добавьте в компонент `AJAXLoader` статический метод `loadHTML`, загружающий HTML-файл с указанным *интернет-адресом* и выводящий его в заданном *элементе*. Метод должен вызываться в формате:

  `loadHTML(<интернет-адрес>, <элемент для вывода>)`

  Сделайте так, чтобы *элемент* можно было указывать либо в виде ссылки на него, либо в виде якоря.

- Переделайте страницу 17.2.html так, чтобы находящийся в ее составе сценарий использовал метод `loadHTML` компонента `AJAXLoader`.

- Переделайте написанную в *уроке 14* страницу 14.5.html таким образом, чтобы данные для вывода диаграммы загружались из файла 17.5.json.

Урок 18
Серверные веб-приложения. Платформа PHP

Серверные веб-приложения
Введение в PHP
Серверные веб-страницы
Разработка PHP-приложений
Отправка данных PHP-приложениям

18.1. Серверные веб-приложения. Платформа PHP

На этом уроке мы отвлечемся от JavaScript и веб-сценариев и будем изучать серверные веб-приложения и их программирование.

> *Серверное веб-приложение* — программа, хранящаяся и исполняющаяся на стороне веб-сервера. В качестве результата выполнения генерирует обычные веб-страницы или данные в формате JSON. Может получать информацию, введенную пользователем в веб-форму, взаимодействовать с базами данных и другими серверными приложениями.

В самом простом случае серверные приложения генерируют веб-страницы, которые открываются веб-обозревателем так же, как обычные, хранящиеся в HTML-файлах. Часто бóльшая часть сайта (а иногда и весь сайт целиком) представляет собой набор серверных приложений.

Серверное веб-приложение хранится на диске серверного компьютера в обычном файле — текстовом или двоичном (в зависимости от программной платформы, на которой оно написано). Обращение к серверному приложению выполняется по интернет-адресу хранящего его файла (так же, как и к любому файлу, входящему в состав сайта).

> *Получив интернет-адрес файла с серверным приложением*, веб-сервер извлекает из него путь к файлу, находит сам файл, но не отправляет его веб-обозревателю, а *запускает на исполнение*. Когда приложение закончит работу, веб-сервер *пересылает клиенту сгенерированную приложением страницу*.

В настройках веб-сервера учтены все возможные типы файлов и записано, какие из них должны пересылаться клиентам, а какие — запускаться на исполнение.

Серверные веб-приложения пишутся на языках Python, Ruby, Perl, даже JavaScript (с использованием исполняющей среды Node.js), не говоря уже о разнообразных компилируемых языках. Но наибольшей популярностью в настоящее время пользуется платформа PHP.

> *PHP* (акроним от PHP: Hypertext Preprocessor, PHP: препроцессор гипертекста) — программная платформа для разработки серверных веб-приложений на одноименном языке программирования.

Серверные приложения, написанные на PHP, хранятся в обычных текстовых файлах в кодировке UTF-8 с расширением php.

Язык PHP весьма прост, очень похож на JavaScript и включает все необходимые программные инструменты.

Все серверные веб-приложения, рассматриваемые в этой книге, написаны на платформе PHP.

Для успешной работы серверных приложений, написанных на PHP, на серверном компьютере должна быть установлена исполняющая среда PHP, а веб-сервер должен быть соответствующим образом настроен.

В состав пакета хостинга XAMPP входит исполняющая среда PHP, а комплектный веб-сервер Apache HTTP Server уже сконфигурирован нужным образом (установка XAMPP описывается в *приложении 2*).

18.2. Упражнение. Изучаем основы PHP

На этом уроке мы напишем на PHP простое серверное приложение, выводящее страницу с текущей датой, и попутно изучим основы PHP-программирования.

> Поскольку языки PHP и JavaScript очень похожи, изучая PHP, мы будем обращать внимание на его отличия от JavaScript.

1. Создадим в корневой папке веб-сервера из пакета XAMPP файл 18.2.php и запишем в него следующий код:

```
<!doctype html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Вывод текущей даты</title>
    </head>
    <body>
        <h1>Текущая дата</h1>
        <p>Сегодня 01.01.2019</p>
    </body>
</html>
```

Пока что это обычная веб-страница, в HTML-код которой мы будем добавлять фрагменты PHP-кода.

2. Сделаем так, чтобы в абзаце с текстом «Сегодня 01.01.2019» вместо 1 января 2019 года выводилась текущая дата. Для этого сначала следует получить эту дату.

   Вставим перед абзацем с текстом «Сегодня 01.01.2019» фрагмент PHP-кода, который получит текущую дату (здесь и далее добавленный и исправленный код выделен полужирным шрифтом):

```
<h1>Текущая дата</h1>
<?php
    $timestamp = time();
?>
<p>Сегодня 01.01.2019</p>
```

   > Фрагмент PHP-кода, помещенный в HTML-код, должен быть заключен в тег `<?php . . . ?>`.

   > Имена переменных в PHP обязательно должны начинаться с символа доллара (`$`).

   > Явно объявлять переменные в PHP не нужно. Для создания переменной достаточно присвоить ей значение.

   Функция `time` возвращает временну́ю отметку с текущими датой и временем в виде количества секунд, прошедших с 1 января 1970 года (временна́я отметка в стиле UNIX). Перед выводом ее следует преобразовать в строку, содержащую дату в привычном нам формате.

3. Добавим в только что написанный PHP-код выражение, выполняющее это преобразование:

```
<?php
    $timestamp = time();
    $date = date('d.m.Y', $timestamp);
?>
```

   Функция `date` преобразует временну́ю отметку в стиле UNIX из второго параметра в строку формата, указанного первым параметром, и возвращает полученную строку в качестве результата. Заданная в первом параметре строка `'d.m.Y'` обозначает формат даты *<число>.<№ месяца>.<год из четырех цифр>*.

   > В PHP все строковые литералы обрабатываются только в строках, взятых в двойные кавычки. В строках, что взяты в одинарные кавычки, обрабатываются лишь литералы `\'` (обозначает одинарную кавычку) и `\\` (обратный слеш).

4. Заменим в абзаце «Сегодня 01.01.2019» дату 1 января 2019 года фрагментом PHP-кода, выводящим строку с текущей датой из переменной `$date`:

```
<p>Сегодня <?php echo $date ?></p>
```

> Оператор вывода `echo`:
>
> `echo <выводимое значение>`
>
> помещает *выводимое значение* в HTML-код генерируемой страницы (как говорят PHP-программисты, ***выполняет вывод***) в том месте, в котором он поставлен.

> В последнем выражении PHP-кода, стоящем непосредственно перед закрывающим тегом `?>`, ставить завершающую точку с запятой необязательно.

5. Запустим веб-сервер из состава XAMPP и в веб-обозревателе выполним переход по интернет-адресу **http://localhost/18.2.php**. Мы увидим обычную веб-страницу, на которой выведена текущая дата (рис. 18.1).

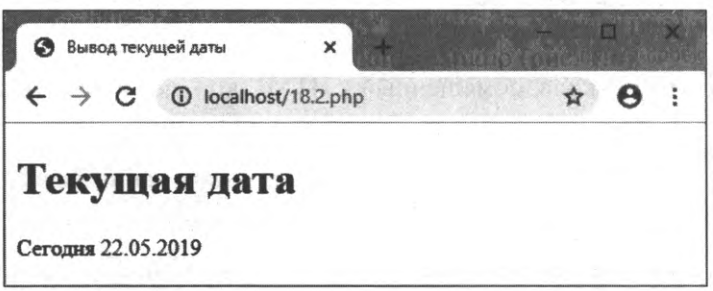


Рис. 18.1. Обычная веб-страница, на которой выведена текущая дата

6. Доработаем серверное приложение, чтобы оно выводило еще и название текущего времени года.
7. Вставим после абзаца «Сегодня...» фрагмент PHP-кода, получающий порядковый номер месяца из текущей временно́й отметки:

```
<p>Сегодня <?php echo $date ?></p>
<?php
    $d = getdate($timestamp);
    $month = $d['mon'];
?>
```

Функция `getdate` возвращает ассоциативный массив с различными составными частями переданной ей временно́й отметки: годом, месяцем, числом и т. д.

> *Ассоциативный массив* (*хеш*) — массив PHP, в котором для доступа к элементам используются не целочисленные, а строковые индексы (*ключи*).

Порядковый номер месяца хранится в элементе с ключом `mon` возвращенного ассоциативного массива.

8. Добавим в тот же фрагмент множественное условное выражение, определяющее время года по номеру месяца:

```
<?php
    $d = getdate($timestamp);
    $month = $d['mon'];
```

```
    if ($month == 1 || $month == 2 || $month == 12)
        $season = 'зима';
    else if ($month >= 3 && $month <= 5)
        $season = 'весна';
    else if ($month >= 6 && $month <= 8)
        $season = 'лето';
    else
        $season = 'осень';
?>
```

9. Добавим туда же выражение, выводящее название времени года:

```
<?php
    . . .
    echo '<p>Сейчас ' . $season . '</p>'
?>
```

Оператор вывода `echo` может выводить и HTML-код, который также будет вставлен в то место страницы, где присутствует этот оператор.

Точка (.) — это оператор конкатенации строк в PHP.

10. Обновим открытую в веб-обозревателе страницу и посмотрим, какое сейчас время года (рис. 18.2).

Важная особенность платформы PHP — в серверных приложениях можно смешивать HTML- и PHP-код.

Серверная веб-страница — страница, содержащая, помимо HTML-кода, еще и фрагменты на языке PHP, исполняемые на стороне сервера.

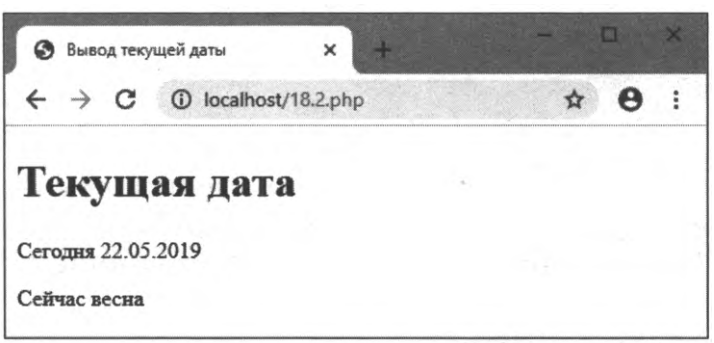


Рис. 18.2. Обновленная веб-страница, показывающая вместе с текущей датой еще и время года

18.3. Упражнение. Пишем простейшую фотогалерею на PHP

Напишем на PHP простейшую фотогалерею, выводящую набор всех изображений, хранящихся в папке. Для простоты сделаем так, чтобы фотогалерея обрабатывала только файлы с расширением jpg.

Найдем в папке 18\!sources сопровождающего книгу электронного архива (см. *приложение 3*) файл 18.3.css и папку images. Скопируем их в корневую папку веб-сервера из состава пакета XAMPP. Файл 18.3.css содержит таблицу стилей, оформляющую фотогалерею, а папка images — графические файлы с изображениями, которые будут выводиться на ее странице.

1. Создадим в корневой папке веб-сервера файл 18.3.php и запишем в него начальный HTML-код:

```
<!doctype html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Фотогалерея</title>
        <link href="18.3.css" rel="stylesheet" type="text/css">
    </head>
    <body>
        <h1>Фотогалерея</h1>
        <section id="photogallery">
        </section>
    </body>
</html>
```

Изображения станем выводить в теге `<section>` с якорем `photogallery`. Каждое изображение будет представлять гиперссылку, указывающую на графический файл и содержащую тег `<img>`.

2. Вставим в элемент `photogallery` пустой тег `<?php . . . ?>` (добавленный код здесь и далее выделен полужирным шрифтом):

```
<section id="photogallery">
    <?php
    ?>
</section>
```

В этом теге мы напишем весь PHP-код, необходимый для вывода изображений из папки images.

3. Напишем выражения, вычисляющие полный путь к этой папке:

```
<?php
    $images_folder = '/images/';
    $current_dir = dirname($_SERVER['SCRIPT_FILENAME']);
    $path = $current_dir . $images_folder;
?>
```

Элемент с ключом `SCRIPT_FILENAME` ассоциативного массива `$_SERVER`, созданного самой платформой PHP, хранит полный путь к текущему PHP-файлу. «Пропустив» его через функцию `dirname`, мы получим только сам путь, без имени файла. А добавив к нему строку `'/images/'` из переменной `$images_folder`, сформируем в переменной `$path` полный путь к нужной папке.

Добавим выражение, открывающее папку images для перебора имеющихся в ней файлов:

```
<?php
    . . .
    if ($dh = opendir($path)) {
    }
?>
```

Функция `opendir` открывает папку, чей путь указан в параметре, и возвращает особый идентификатор или `false`, если папку открыть не удалась. Мы присваиваем возвращенный ей результат переменной `$dh` и сразу же проверяем, не равен ли он `false`. И в PHP, и в JavaScript оператор присваивания возвращает присвоенное значение, следовательно, его можно использовать в качестве условия в условном выражении.

4. Впишем в условное выражение код, перебирающий файлы в открытой папке:

```
<?php
    . . .
    if ($dh = opendir($path)) {
        while (($file = readdir($dh)) !== false){
            if ($file != '.' && $file != '..') {
                echo '<a href="">';
                echo '<img src="' . $images_folder . $file . '">';
                echo '</a>';
            }
        }
    }
?>
```

Функция `readdir` возвращает имя очередного файла из открытой папки, идентификатор которой передан с параметром, или `false`, если файлов больше нет. Мы перебираем в цикле файл за файлом и формируем HTML-теги, необходимые для вывода этого файла в фотогалерее.

Следует учесть, что первыми двумя файлами, возвращенными этой функцией, станут файл с именами . (точка), представляющий текущую папку, и файл .. (две точки), представляющий папку, в которую вложена текущая. Их выводить в фотогалерее не следует.

5. Добавим выражение, которое закроет папку images по завершении перебора файлов:

```
<?php
    . . .
    if ($dh = opendir($path)) {
        while (($file = readdir($dh)) !== false){
            . . .
        }
```

```
        closedir($dh);
    }
?>
```

Функция `closedir` закрывает папку с указанным в параметре идентификатором.

6. Запустим встроенный в XAMPP веб-сервер и перейдем в веб-обозревателе по интернет-адресу **http://localhost/18.3.php**. Мы увидим страницу с фотогалереей (рис. 18.3).

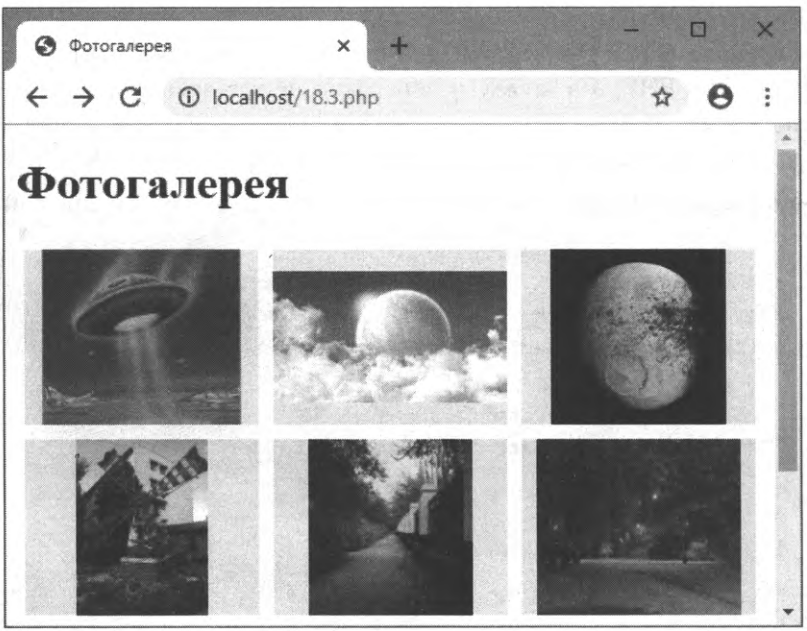


Рис. 18.3. Веб-страница с фотогалереей

Каждый раз, открывая страницу, мы набираем в составе интернет-адреса имя файла — например, **http://localhost/18.3.php** (здесь это имя для ясности подчеркнуто). Это надоедает и часто приводит к ошибкам в наборе. Исключим имя файла из интернет-адреса, превратив 18.3.php в страницу по умолчанию.

> *Веб-страница по умолчанию* — страница, к которой веб-сервер обращается, если путь к запрошенному файлу не содержит имени файла и фактически указывает на папку.
>
> Файл веб-страницы по умолчанию должен иметь имя index и расширение html (если это обычная страница) или php (если серверная).

7. Переименуем файл 18.3.php в index.php, превратив его в страницу по умолчанию.

В веб-обозревателе выполним переход по более короткому интернет-адресу **http://localhost/** и проверим, открывается ли страница.

18.4. Упражнение. Передаем данные методом GET

Если щелкнуть на любой картинке из нашей фотогалереи, ничего не произойдет. Реализуем по щелчку просмотр полных редакций изображений на отдельных страницах с выводом текстовых описаний (если они есть).

Для вывода страниц с изображениями напишем серверную страницу get.php. Имя файла с выводимым изображением будет передаваться ей страницей index.php методом GET под именем `image`. Описания будут храниться в папке descriptions в текстовых файлах с теми же именами, что и у файлов с изображениями.

Для передачи методом GET отдельные величины:

- кодируются — в частности, пробелы преобразуются в символы + (плюс), а буквы кириллицы и знаки препинания — в их числовые коды;
- сводятся в пары вида:

  <имя передаваемой величины>=<ее значение>;
- пары объединяются в одну строку и отделяются друг от друга амперсандами (`&`);
- строка добавляется в конец интернет-адреса целевой страницы и отделяется от него вопросительным знаком (`?`).

> Отдельные величины, передаваемые методом GET, называются *GET-параметрами*.

Вот пример передачи странице get.php методом GET параметра `image` со значением `'20190402114802'` и параметра `mode` со значением `'show'`:

/get.php?image=20190402114802&mode=show

> В PHP-коде все полученные GET-параметры хранятся в ассоциативном массиве `$_GET`, создаваемом самой платформой. Ключи элементов этого массива совпадают с именами GET-параметров.

Вот пример извлечения значения GET-параметра `image`:

```
$fn = $_GET['image'];
```

1. Итак, чтобы реализовать задуманное, в папке 18\!sources сопровождающего книгу электронного архива (см. *приложение 3*) найдем папку descriptions и скопируем в корневую папку встроенного в пакет XAMPP веб-сервера.
2. Откроем страницу index.php, найдем код, создающий фотогалерею, и внесем в него следующие правки (добавленный и исправленный код здесь и далее выделен полужирным шрифтом):

```
if ($file != '.' && $file != '..') {
    $n = pathinfo($file, PATHINFO_FILENAME);
    echo '<a href="/get.php?image=' . $n . '" id="' . $n . '">';
    . . .
}
```

Чтобы получить имя файла без расширения, мы воспользовались функцией pathinfo. Она принимает первым параметром путь к файлу и возвращает его часть, указанную во втором параметре. Константа PATHINFO_FILENAME обозначает имя файла без расширения.

В каждой гиперссылке дополнительно формируется якорь, совпадающий со значением GET-параметра image. Он понадобится нам позже, при реализации возврата.

3. Создадим в корневой папке веб-сервера файл get.php и запишем в него начальный HTML-код:

```
<!doctype html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Изображение</title>
        <link href="18.3.css" rel="stylesheet" type="text/css">
    </head>
    <body>
        <section id="photo">
        </section>
        <p><a href="">Назад</a></p>
    </body>
</html>
```

В теге <section> с якорем photo выведем тег <img> с нужным изображением, а над элементом photo, в виде заголовка первого уровня, — описание к этому изображению.

4. Вставим перед тегом <section> PHP-код, который вычислит полный путь к текстовому файлу с описанием и подготовит другие нужные данные:

```
<?php
    $images_folder = '/images/';
    $descriptions_folder = '/descriptions/';
    $current_dir = dirname($_SERVER['SCRIPT_FILENAME']);
    $fn = $_GET['image'];
    $path = $current_dir . $descriptions_folder . $fn . '.txt';
?>
<section id="photo">
</section>
```

5. Добавим к этому коду фрагмент, считывающий из текстового файла описание и выводящий его на экран:

```
<?php
    . . .
    if (file_exists($path)) {
        $s = file_get_contents($path);
        echo '<h1>' . $s . '</h1>';
```

```
    } else
        echo '<h1>Изображение</h1>';
?>
```

Функция `file_exists` возвращает `true`, если файл с указанным в параметре путем существует. Функция `file_get_contents` считывает содержимое файла и возвращает его в виде строки.

Если текстовый файл с описанием отсутствует, на странице вводится заголовок «Изображение».

6. Вставим в тег `<section>` с якорем `photo` PHP-код, выводящий само изображение:

```
<section id="photo">
    <?php
        $path = $images_folder . $fn . '.jpg';
        echo '<img src="' . $path . '">'
    ?>
</section>
```

7. Вставим в тег гиперссылки **Назад** PHP-код, задающий целевой интернет-адрес:

```
<p><a href="/#<?php echo $fn ?>">Назад</a></p>
```

Это интернет-адрес страницы с фотогалереей, к которому в качестве якоря добавлено имя просматриваемого изображения. В результате при возврате на страницу фотогалереи ее содержимое будет прокручено до элемента с указанным в адресе якорем — до гиперссылки с миниатюрой просматриваемого изображения. Так мы реализуем корректный возврат на то же место страницы, с которого ушли.

Запустим веб-сервер, откроем страницу фотогалереи, пощелкаем на разных миниатюрах и проверим, выводятся ли изображения и описания к ним (рис. 18.4).

Рис. 18.4. Щелчки на миниатюрах фотогалереи выводят полноразмерные изображения и описания к ним

18.5. Упражнение. Передаем данные методом POST

Постоянно любоваться на одни и те же картинки скоро надоест. Поэтому добавим средства для загрузки в нашу фотогалерею новых изображений.

Этим займется серверная страница app.php, где мы создадим веб-форму с полем выбора графического файла и областью редактирования, куда будет заноситься описание. PHP-код, сохраняющий выгруженный файл и описание к нему, мы для удобства поместим в ту же страницу app.php. А пересылать данные из веб-формы станем методом POST.

Веб-форма пересылает введенные в нее данные самостоятельно — для этого необходимо лишь нажать кнопку отправки данных. Данные могут быть отправлены как методом POST, так и методом GET.

> В PHP-коде данные, переданные методом POST (*POST-параметры*), хранятся в автоматически создаваемом ассоциативном массиве `$_POST`.

> Выгруженные файлы в PHP хранятся в ассоциативном массиве `$_FILES`. Каждый его элемент также представляет собой ассоциативный массив, хранящий различные сведения о полученном файле (в частности, «временное» имя, размер и тип).

1. Создадим в корневой папке веб-сервера файл add.php с таким начальным HTML-кодом:

```
<!doctype html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Добавление изображения</title>
        <link href="18.3.css" rel="stylesheet" type="text/css">
    </head>
    <body>
        <h1>Добавление изображения</h1>
        <form action="/add.php" method="post"
        enctype="multipart/form-data">
            <p>Изображение:<br><input type="file"
            name="file" accept=".jpg" required></p>
            <p>Описание:<br><textarea
            name="description"></textarea></p>
            <p><input type="submit" name="submit"
            value="Добавить"></p>
        </form>
        <p><a href="/">Назад</a></p>
    </body>
</html>
```

В форме мы создали:

- поле выбора файла `file`, принимающее только файлы с расширением jpg и обязательное к заполнению, — для указания файла с добавляемым изображением;
- область редактирования `description` — для занесения описания;
- кнопку отправки данных `submit`.

Для успешной отправки файлов данные, занесенные в веб-форму, должны быть закодированы методом `multipart/form-data` (указан в атрибуте `enctype` тега `<form>`).

Данные из формы на странице add.php будут отправлены для обработки и сохранения той же самой странице — add.php (см. значение атрибута `action` тега `<form>`). Это значит, что при добавлении нового изображения эта страница будет загружена и обработана дважды:

- первый раз — когда пользователь переходит на нее, намереваясь добавить в галерею изображение;
- второй раз — когда пользователь нажимает кнопку отправки данных, чтобы сохранить указанное им в форме изображение в галерее.

Определить, в какой раз открывается страница, мы можем, проверив существование в ассоциативном массиве `$_POST` какого-либо POST-параметра — например, `submit` (он соответствует кнопке отправки данных `submit`). Если он в массиве есть, мы сохраняем полученный файл и выполняем перенаправление на страницу галереи, чтобы пользователь сразу увидел добавленное изображение.

2. Поместим в самом начале HTML-кода фрагмент на языке PHP, который проверит, во второй ли раз была загружена страница (добавленный код здесь и далее выделен полужирным шрифтом):

```
<?php
    if (isset($_POST['submit'])) {
    }
?>
<!doctype html>
. . .
```

Функция `isset` возвращает `true`, если ей была передана существующая переменная или элемент массива.

3. Поместим в только что написанное условное выражение код, вычисляющий пути к папкам images и descriptions:

```
<?php
    if (isset($_POST['submit'])) {
        $images_folder = '/images/';
        $descriptions_folder = '/descriptions/';
        $current_dir = dirname($_SERVER['SCRIPT_FILENAME']);
```

```
        $images_path = $current_dir . $images_folder;
        $descriptions_path = $current_dir . $descriptions_folder;
    }
?>
```

Внимание!

Сохранять выгруженный файл в PHP под его изначальным именем не рекомендуется — если в имени присутствуют символы кириллицы, они будут преобразованы в нечитаемые последовательности символов. Это происходит из-за ошибки в программном ядре платформы, не исправленной до сих пор.

4. Мы будем сохранять выгруженные файлы под именами вида:

   <год><№ месяца><число><часы><минуты><секунды>

   Допишем код, который будет формировать такое имя файла:

```
<?php
    if (isset($_POST['submit'])) {
        . . .
        $d = getdate(time());
        $file_base_name = $d['year'] . $d['mon'] . $d['mday'] .
            $d['hours'] . $d['minutes'] . $d['seconds'];
    }
?>
```

   Вызовом функции `time` мы получаем текущую временну́ю отметку в стиле UNIX, а «пропустив» ее через функцию `getdate`, вызванную без второго параметра, — ассоциативный массив с различными частями даты и времени. Далее мы извлекаем элементы этого массива и составляем из них имя файла.

   PHP временно сохраняет все выгруженные файлы в особой папке под «временными» именами. «Временное» имя каждого файла хранится в элементе с ключом `tmp_name` ассоциативного массива, содержащего сведения о файле. Все эти файлы должны быть перемещены на место постоянного хранения.

5. Добавим код, формирующий имя для выгруженного файла с изображением и сохраняющий его в папке images:

```
<?php
    if (isset($_POST['submit'])) {
        . . .
        $image_file_name = $images_path . $file_base_name . '.jpg';
        move_uploaded_file($_FILES['file']['tmp_name'],
            $image_file_name);
    }
?>
```

   Функция `move_uploaded_file` сохраняет полученный из формы файл, чье «временное» имя указано в первом параметре, по пути из второго параметра.

6. Теперь следует проверить, занес ли пользователь описание для картинки, и если это так, сохранить его в текстовом файле с тем же именем в папке descriptions.

7. Добавим код, сохраняющий описание (если оно есть):

```
<?php
    if (isset($_POST['submit'])) {
        . . .
        if (isset($_POST['description']) &&
            !empty($_POST['description'])) {
            $description_file_name = $descriptions_path .
                $file_base_name . '.txt';
            $file = fopen($description_file_name, 'w');
            fwrite($file, $_POST['description']);
            fclose($file);
        }
    }
?>
```

Сначала проверяем, присутствует ли в массиве `$_POST` элемент с ключом `description` и не содержит ли он пустую строку или строку из одних пробелов (нам поможет функция `empty`, возвращающая `true`, если ей была передана такая строка). Далее формируем имя файла с описанием, открываем его на запись (вызвав функцию `fopen` со вторым параметром `'w'`), записываем в него описание (вызовом функции `fwrite`) и закрываем (вызвав функцию `fclose`).

8. Допишем код, выполняющий перенаправление на страницу с фотогалереей:

```
<?php
    if (isset($_POST['submit'])) {
        . . .
        header('Location: /');
    }
?>
```

Функция `header` вставляет в отправляемый клиенту ответ заданный заголовок.

> *Заголовок* — находится в начале серверного ответа (или клиентского запроса), содержит сведения о передаваемых данных или какое-либо указание веб-обозревателю (веб-серверу).

Указание выполнить перенаправление по заданному *интернет-адресу*, отправляемое веб-обозревателю, записывается заголовком следующего формата:
`Location: <целевой интернет-адрес>`

После перенаправления выполнять остальной код серверной страницы ни к чему.

9. Добавим выражение, завершающее работу серверного приложения вызовом функции `exit`:

```
<?php
    if (isset($_POST['submit'])) {
        . . .
        exit();
    }
?>
```

10. Откроем страницу index.php и вставим между заголовком и элементом photogallery гиперссылку на страницу добавления изображения:

```
<h1>Фотогалерея</h1>
<p><a href="/add.php">Добавить изображение</a></p>
<section id="photogallery">
    . . .
</section>
```

Запустим веб-сервер, откроем страницу фотогалереи, перейдем на страницу добавления изображения и добавим в галерею несколько новых картинок.

> По умолчанию PHP позволяет отправлять на сервер файлы размером не более 2 Мбайт. Увеличить максимальный допустимый размер пересылаемого файла можно в настройках PHP (см. *приложение 2*).

18.6. Самостоятельные упражнения

- Сделайте так, чтобы в качестве названия (задается в теге <title>) страницы get.php выводилось описание просматриваемого изображения, если оно есть.
- Реализуйте удаление изображений.

  Подсказка:

  - поместите на странице get.php веб-форму со скрытым полем name, хранящим имя выведенного изображения (то самое, что передается этой странице с GET-параметром image), и кнопкой отправки данных submit;
  - добавьте в начало той же страницы PHP-код, срабатывающий, если в составе полученных POST-параметров есть параметр submit;
  - для удаления файла используйте функцию unlink:

    unlink(*<путь к удаляемому файлу>*)

  - не забудьте, что нужно также удалить файл с описанием, если он есть;
  - после удаления выполните перенаправление на страницу с фотогалереей.

Урок 19
Разработка фронтендов и бэкендов

Фронтенды и бэкенды
Веб-службы
Отправка данных веб-службам средствами AJAX

19.1. Веб-разработка: старый и новый подходы. Фронтенды и бэкенды. Веб-службы

В настоящее время в веб-программировании находят применение два подхода: старый и новый.

- *Старый подход* — серверные приложения генерируют полноценные веб-страницы, обрабатываемые непосредственно веб-обозревателями. Применяется при разработке традиционных сайтов (см. *урок 18*).
- *Новый подход* — серверные приложения генерируют фрагменты HTML-кода или данные в формате JSON. Загрузка, обработка и вывод таких данных выполняются веб-сценариями (часто — очень сложными) с применением технологии AJAX. Применяется при программировании сверхдинамических сайтов (см. *разд. 17.2*).

Новый подход имеет перед старым ряд преимуществ:

- повышение быстродействия — поскольку пересылка по Сети компактных HTML-фрагментов и JSON-файлов выполняется значительно быстрее, нежели доставка объемных веб-страниц;
- уменьшение нагрузки на серверный компьютер — поскольку генерирование HTML-фрагментов и JSON-файлов отнимает меньше системных ресурсов;
- возможность повторного использования полученных данных — одни и те же JSON-данные могут быть выведены на странице в разных форматах (например, в виде краткого списка и подробной таблицы).

Сайт, написанный с применением нового подхода, состоит из двух частей: фронтенда и бэкенда.

> *Фронтенд* — вся совокупность клиентских веб-приложений, работающих в веб-обозревателе и взаимодействующих с пользователем.

Бэкенд — вся совокупность серверных веб-служб, работающих совместно с веб-сервером и обслуживающих фронтенд.

Веб-служба — серверное веб-приложение, генерирующее данные в формате JSON или фрагменты HTML-кода.

19.2. Упражнение. Новая фотогалерея. Вывод списка миниатюр

Напишем версию фотогалереи, основанную на новом подходе и состоящую из фронтенда и бэкенда.

1. В папке 19\!sources сопровождающего книгу электронного архива (см. *приложение 3*) найдем папки images, descriptions, файлы styles.css и ajaxloader.js и скопируем их в корневую папку веб-сервера из пакета XAMPP. В папках images и descriptions хранятся, соответственно, сами изображения и описания к ним, в файле styles.css — таблица стилей для оформления фотогалереи, а в файле ajaxloader.js — код компонента `AJAXLoader`, который мы написали в *разд. 17.6* и который сейчас используем для загрузки данных.
2. Начнем с написания бэкенда — веб-службы get.php, которая будет искать файлы в папке images и пересылать фронтенду их перечень в виде JSON-нотации объекта со свойством `data`, хранящим массив объектов со свойствами `path` (ссылка на очередной файл) и `id` (идентификатор файла — его имя без расширения).

   Создадим в корневой папке файл get.php и сохраним в нем такой код:

```
<?php
$images_folder = '/images/';
$current_dir = dirname($_SERVER['SCRIPT_FILENAME']);
$output['data'] = [];
$path = $current_dir . $images_folder;
if ($dh = opendir($path)) {
    while (($file = readdir($dh)) !== false){
        if ($file != '.' && $file != '..') {
            $el['path'] = $images_folder . $file;
            $el['id'] = pathinfo($file, PATHINFO_FILENAME);
            $output['data'][] = $el;
        }
    }
    closedir($dh);
}
$json = json_encode($output);
echo $json;
?>
```

   В переменной `$output` создаем ассоциативный массив, содержащий элемент с ключом `data`, в котором и сохраняем массив с файлами. Аналогичным образом формируем ассоциативные массивы, хранящие сведения об отдельных файлах.

Для добавления элемента в массив применяем поддерживаемую PHP конструкцию вида:

```
<переменная с массивом>[] = <значение нового элемента>;
```

Для кодирования готового массива в формат JSON используем функцию `json_encode`. Она корректно преобразует ассоциативные массивы PHP в JSON-нотации объектов.

Чтобы отправить закодированные в формат JSON данные клиенту, применяем знакомый оператор вывода `echo`.

3. Проверим, как работает готовая веб-служба. Запустим веб-сервер и в веб-обозревателе перейдем по интернет-адресу **http://localhost/get.php**. Мы увидим перечень файлов в формате JSON (рис. 19.1).

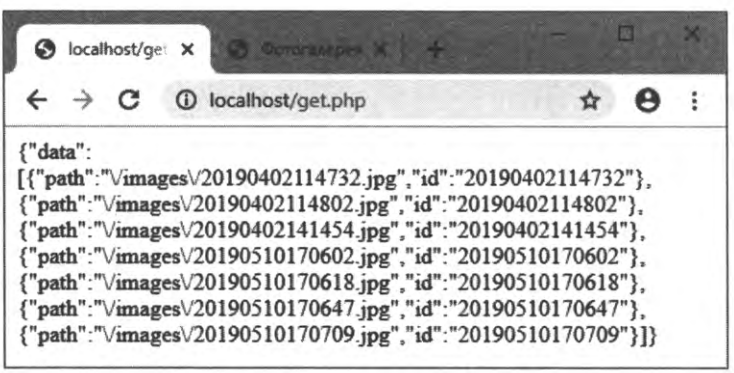


Рис. 19.1. Перечень файлов в формате JSON

4. Теперь можно приступить к программированию фронтенда.

Напишем компонент `PGList`, загружающий и выводящий список изображений, сохранив его код в файле сценария pglist.js. Для загрузки списка изображений применим компонент `AJAXLoader`. Этот и все последующие компоненты, которые мы напишем, будут формировать программно все содержимое страницы. Следовательно, в качестве базового элемента для них мы можем использовать секцию тела страницы (тег `<body>`).

Создадим в корневой папке файл pglist.js с кодом компонента `PGList`:

```
const PGList = (function () {
    function show(response) {
        const data = JSON.parse(response).data;
        document.body.innerHTML = '';
        let h1 = document.createElement('h1');
        h1.textContent = 'Фотогалерея';
        document.body.appendChild(h1);
        let section = document.createElement('section');
        section.id = 'photogallery';
        let a, img;
```

```
        data.forEach((el) => {
            a = document.createElement('a');
            img = document.createElement('img');
            img.src = el.path;
            a.appendChild(img);
            section.appendChild(a);
        });
        document.body.appendChild(section);
        document.title = 'Фотогалерея';
        window.scrollTo(0, 0);
    }
    function PGList() {
        this.ajax = new AJAXLoader();
        this.ajax.load('get.php', show);
    }
    return PGList;
})();
```

Обычно конструктор компонента принимает в качестве первого параметра базовый элемент. Но в нашем случае это не нужно, поскольку базовым элементом всегда будет выступать секция тела страницы.

Конструктор сразу же создаст объект компонента `AJAXLoader` и запустит загрузку данных, формируемых веб-службой get.php.

Функция `show` выполнится после получения данных, создаст заголовок страницы, список миниатюр, задаст для страницы название **Фотогалерея** и прокрутит ее содержимое в самый верх.

5. Закончив компонент, приступим к работе над одной-единственной страницей нашего сверхдинамического сайта — index.html — и создадим в корневой папке файл index.html с таким кодом:

```
<!doctype html>
<html>
    <head>
        <meta charset="UTF-8">
        <title></title>
        <link href="styles.css" rel="stylesheet" type="text/css">
        <script src="ajaxloader.js" type="text/javascript"></script>
        <script src="pglist.js" type="text/javascript"></script>
    </head>
    <body>
    </body>
</html>
<script type="text/javascript">
    let component = new PGList();
</script>
```

Не забываем привязать таблицу стилей styles.css, файлы сценария ajaxloader.js и pglist.js. Поскольку название и все содержание страницы будут формироваться программно, оставим теги `<title>` и `<body>` пустыми.

В веб-обозревателе перейдем по интернет-адресу **http://localhost/** — и увидим нашу сверхдинамическую фотогалерею (рис. 19.2).

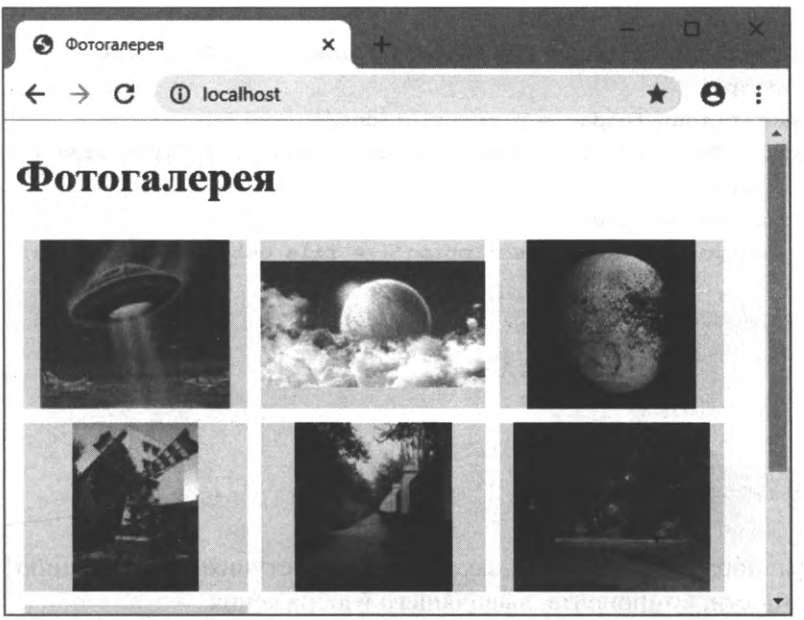


Рис. 19.2. Сверхдинамическая фотогалерея, основанная на новом подходе и состоящая из фронтенда и бэкенда

19.3. Упражнение. Новая фотогалерея. Показ изображений

Сделаем так, чтобы при щелчке на миниатюре выводилась полная редакция изображения с описанием (если таковое есть).

Средства для выдачи сведений о запрашиваемом изображении встроим в написанную ранее веб-службу get.php. При этом:

- если эта веб-служба получит GET-параметр `image` с идентификатором файла, она вернет сведения об этом файле — JSON-нотацию объекта со свойством `data`, хранящим объект со свойствами `path` (ссылка на файл) и `description` (описание, может отсутствовать);
- если веб-служба не получит GET-параметр `image`, она, как и ранее, выдаст перечень файлов с изображениями.

* * *

1. Откроем веб-службу get.php и допишем необходимый код (выделен полужирным шрифтом):

```
<?php
$images_folder = '/images/';
$current_dir = dirname($_SERVER['SCRIPT_FILENAME']);
$output['data'] = [];
if (isset($_GET['image'])) {
    $output['data']['path'] = $images_folder . $_GET['image'] .
        '.jpg';
    $descriptions_folder = '/descriptions/';
    $path = $current_dir . $descriptions_folder . $_GET['image'] .
        '.txt';
    if (file_exists($path))
        $output['data']['description'] = file_get_contents($path);
} else {
    $path = $current_dir . $images_folder;
    if ($dh = opendir($path)) {
        . . .
    }
}
. . .
?>
```

2. На этом доработки бэкенда закончены. Приступим к написанию фронтенда и, в частности, компонента, выводящего изображения.

   Вывод изображения будет выполняться по щелчку на гиперссылке в списке миниатюр (он выводится компонентом `PGList`). Компоненту, который выведет изображение, нужно передать его идентификатор. Будем передавать его через интернет-адрес гиперссылки, записав в виде якоря такого формата:

   `get=<идентификатор изображения>`

   Итак, откроем файл pglist.js и дополним код компонента `PGList`:

```
const PGList = (function () {
    function show(response) {
        . . .
        data.forEach((el) => {
            a = document.createElement('a');
            a.href = '#get=' + el.id;
            img = document.createElement('img');
            . . .
        });
        . . .
    }
    . . .
})();
```

3. Напишем компонент, который будет выводить изображение с идентификатором, полученным с первым параметром. Дадим ему имя `PGItem` и сохраним в файле сценария pgitem.js.

   Для этого создадим в корневой папке файл pgitem.js и введем в него код:

```
const PGItem = (function () {
    function show(response) {
        const data = JSON.parse(response).data;
        const desc = data.description || 'Изображение';
        document.body.innerHTML = '';
        let h1 = document.createElement('h1');
        h1.textContent = desc;
        document.body.appendChild(h1);
        let section = document.createElement('section');
        section.id = 'photo';
        let img = document.createElement('img');
        img.src = data.path;
        section.appendChild(img);
        document.body.appendChild(section);
        let p = document.createElement('p');
        let a = document.createElement('a');
        a.href = '';
        a.textContent = 'Назад';
        p.appendChild(a);
        document.body.appendChild(p);
        document.title = desc;
        window.scrollTo(0, 0);
    }
    function PGItem(id) {
        this.ajax = new AJAXLoader();
        this.ajax.load('get.php?image=' + id, show);
    }
    return PGItem;
})();
```

   В конструкторе сразу же запускается загрузка сведений об изображении с указанным в параметре идентификатором. После загрузки выполнится функция `show` — она выведет заголовок страницы, изображение и гиперссылку **Назад** с «пустым» интернет-адресом.

4. Откроем файл страницы index.html и добавим в секцию заголовка (тег `<head>`) тег `<script>`, привязывающий файл pgitem.js:

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

5. При щелчках на гиперссылках фактически будет выполняться переход по якорям. Выяснять, по какому якорю был выполнен переход, и активизировать нужный компонент удобнее в обработчике события `hashchange` окна. При этом:
   - если якорь записан в формате: `get=<идентификатор изображения>`, нужно создать объект компонента `PGItem`, передав ему полученный идентификатор;
   - если якорь «пуст» — нужно создать объект компонента `PGList`.

   Удалим весь JavaScript-код, имеющийся в составе страницы index.html, и заменим его кодом, выполняющим маршрутизацию:

```
let component;
function route() {
    let hash = location.hash;
    const getRE = /get=(\d+)/i;
    let result;
    if (result = getRE.exec(hash))
        component = new PGItem(result[1]);
    else if (!hash)
        component = new PGList();
}
window.addEventListener('hashchange', route);
route();
```

Рис. 19.3. Полная редакция выбранного изображения

> *Маршрутизация* — вывод на страницу того или иного компонента в зависимости от якоря, по которому был выполнен переход.

Обработчик события `hashchange` окна мы реализовали в виде функции `route`. Для проверки факта щелчка на миниатюре и выделения идентификатора из якоря применили регулярное выражение.

Сразу после открытия страницы событие `hashchange` не возникает. Поэтому в конце кода мы вызвали эту функцию явно.

Запустим веб-сервер и перейдем по интернет-адресу **http://localhost/**. Щелкнем на какой-либо миниатюре — и увидим полную редакцию соответствующего изображения (рис. 19.3).

19.4. Отправка веб-службе данных методом POST

Данные, отправляемые методом POST, указываются в вызове метода `send` объекта AJAX:

```
send(<отправляемые данные>)
```

При этом в предшествующем вызове метода `open` в первом параметре следует указать строку `'POST'`.

Отправить данные методом POST можно тремя способами.

19.4.1. Отправка веб-формы целиком

Если нужно отправить веб-службе все данные, занесенные в форму, можно поступить следующим образом:

- получить DOM-объект формы;
- создать с помощью оператора `new` объект класса `FormData`, указав в качестве параметра полученный ранее объект формы;
- передать методу `send` объекта AJAX только что созданный объект класса `FormData`.

Пример:

```
<form id="frm">
    . . .
</form>
. . .
const frm = document.getElementById('frm');
const fd = new FormData(frm);
const ajax = new XMLHttpRequest();
ajax.open('POST', 'add.php', true);
ajax.send(fd);
```

Данные, отправляемые с помощью класса `FormData`, всегда кодируются с применением метода `multipart/form-data`. Благодаря этому становится возможным отправлять файлы.

19.4.2. Отправка произвольных данных

Для отправки веб-службе методом POST произвольных данных (часть которых, возможно, взята из веб-формы) нужно:

- создать объект класса `FormData` без указания параметров;
- добавить в него все нужные значения, воспользовавшись методом `append` этого объекта:

  `append(<`*имя POST-параметра*`>, <`*значение*`>)`

  Имя POST-параметра указывается в виде строки, *значение*, если не является строкой, будет преобразовано в строку;
- если нужно добавить файлы, применив другой формат вызова метода `append`:

  `append(<`*имя POST-параметра*`>, <`*файл*`>[, <`*имя файла*`>])`

  Файл указывается в виде объекта класса `File` (см. *разд. 13.1*), *имя файла* задается в виде строки. Если же оно не указано, отправляемый файл получит имя `'blob'`;
- передать методу `send` объекта AJAX объект класса `FormData` с добавленными в него данными.

Пример:

```
const description = document.getElementById('description');
const image = document.getElementById('image');
const fd = new FormData();
fd.append('description', description.value);
fd.append('image', image.files[0], image.files[0].name);
fd.append('operation', 'addimage');
const ajax = new XMLHttpRequest();
ajax.open('POST', 'add.php', true);
ajax.send(fd);
```

19.4.3. Отправка данных в виде строки POST-параметров

Если объем отправляемых методом POST данных *невелик и не включает файлы*, можно отправить их в виде строки, для чего:

- закодировать отдельные передаваемые значения, применив встроенную в JavaScript функцию `encodeURIComponent`:

  `encodeURIComponent(<`*кодируемое значение*`>)`

  Закодированное *значение* возвращается в качестве результата.

Кодирование необходимо только в том случае, если *значение* содержит символы, отличные от букв латиницы и цифр;

- свести их в пары вида:

  <имя POST-параметра>=<закодированное значение>

- объединить пары в одну строку и отделить их друг от друга амперсандами (&);
- добавить в отправляемый веб-серверу запрос заголовок вида:

  Content-Type: *<метод кодирования данных>*

  Сделать это можно вызовом метода `setRequestHeader` у объекта AJAX:

  `setRequestHeader(`*<имя заголовка>*`, `*<значение заголовка>*`)`

  В качестве *имени заголовка* следует указать строку `'Content-Type'`, а в качестве значения — строку `'application/x-www-form-urlencoded'` или `'multipart/form-data'`;

- передать методу `send` объекта AJAX получившуюся строку.

Пример:

```
const imageId = document.getElementById('image_id')
const comment = document.getElementById('comment')
// Идентификатор изображения, заданный в поле imageId, содержит
// только цифры, поэтому его кодировать необязательно
let s = 'image=' + imageId.value;
s += '&comment=' + encodeURIComponent(comment.value);
s += '&operation=delete';
const ajax = new XMLHttpRequest();
ajax.open('POST', 'delete.php', true);
ajax.setRequestHeader('Content-Type',
    'application/x-www-form-urlencoded');
ajax.send(s);
```

19.5. Упражнение. Новая фотогалерея. Загрузка изображений в галерею

Реализуем в новой фотогалерее средства для загрузки в нее новых изображений.

Для этого в состав бэкенда добавим веб-службу add.php, которая сохранит загруженный на сервер файл и его описание. Данные она получит в тех же POST-параметрах, что и страница app.php из *разд. 18.5*.

1. Добавим в корневую папку веб-сервера файл add.php с таким кодом:

```
<?php
$images_folder = '/images/';
$descriptions_folder = '/descriptions/';
```

```
$current_dir = dirname($_SERVER['SCRIPT_FILENAME']);
$images_path = $current_dir . $images_folder;
$descriptions_path = $current_dir . $descriptions_folder;
$d = getdate(time());
$file_base_name = $d['year'] . $d['mon'] . $d['mday'] . $d['hours'] .
    $d['minutes'] . $d['seconds'];
$image_file_name = $images_path . $file_base_name . '.jpg';
move_uploaded_file($_FILES['file']['tmp_name'], $image_file_name);
if (isset($_POST['description']) && !empty($_POST['description'])) {
    $description_file_name = $descriptions_path .  $file_base_name .
       '.txt';
    $file = fopen($description_file_name, 'w');
    fwrite($file, $_POST['description']);
    fclose($file);
}
$output['data']['status'] = 1;
$json = json_encode($output);
echo $json;
?>
```

В самом конце мы отсылаем клиенту ответ — JSON-нотацию объекта со свойством `data`, которое хранит объект со свойством `status`, содержащим число `1`. Так мы уведомим клиента, что операция была успешно выполнена.

2. Теперь — фронтенд. Условимся, что компонент, реализующий форму для загрузки изображения (мы напишем его совсем скоро), будет выводиться на экран при переходе по якорю `add`. И сразу же добавим в код компонента `PGList`, выводящего список миниатюр, фрагмент, который создаст гиперссылку **Добавить изображение**.

   Откроем файл pglist.js и внесем в него следующие правки:

```
const PGList = (function () {
    function show(response) {
        . . .
        document.body.appendChild(h1);
        let p = document.createElement('p');
        let a = document.createElement('a');
        a.href = '#add';
        a.textContent = 'Добавить изображение';
        p.appendChild(a);
        document.body.appendChild(p);
        let section = document.createElement('section');
        section.id = 'photogallery';
        let a, img;
        data.forEach((el) => {
            . . .
        });
```

```
        . . .
    }
    . . .
})();
```

3. Теперь добавим в компонент AJAXLoader метод loadPOST, который отправит методом POST данные веб-службе и загрузит результат ее работы. Первым параметром он получит интернет-адрес веб-службы, вторым — отправляемые данные, третьим — функцию, вызываемую после успешной загрузки, необязательным четвертым — функцию, вызываемую в случае неудачи.

   Итак, откроем файл ajaxloader.js и добавим в него объявление метода loadPOST:

```
const AJAXLoader = (function () {
    . . .
    AJAXLoader.prototype.load = function (url, success, error) {
        . . .
    };
    AJAXLoader.prototype.loadPOST = function (url,
        data, success, error) {
        this.ajax.success = success;
        this.ajax.error = error || this.defaultErrorHandler;
        this.ajax.open(AJAXLoader.POST, url, true);
        this.ajax.send(data);
    };
    AJAXLoader.loadHTML = function (url, el) {
        . . .
    };
    return AJAXLoader;
})();
```

4. Напишем компонент PGAdd, который выведет форму для выбора изображения, загрузит изображение на сервер и в случае успеха выполнит перенаправление на список миниатюр.

   Создадим в корневой папке файл pgadd.js с кодом этого компонента:

```
const PGAdd = (function () {
    let __ajax, __file, __description;
    function redirect() {
        location.hash = '';
    }
    function sendData(evt) {
        const fd = new FormData();
        fd.append('file', __file.files[0], __file.files[0].name);
        if (__description.value)
            fd.append('description', __description.value);
        __ajax = new AJAXLoader();
        __ajax.loadPOST('add.php', fd, redirect);
        evt.preventDefault();
    }
```

```
    function PGAdd() {
        document.body.innerHTML = '';
        let h1 = document.createElement('h1');
        h1.textContent = 'Добавление изображения';
        document.body.appendChild(h1);
        let form = document.createElement('form');
        let p = document.createElement('p');
        let txt = document.createTextNode('Изображение');
        p.appendChild(txt);
        let br = document.createElement('br');
        p.appendChild(br);
        __file = document.createElement('input');
        __file.type = 'file';
        __file.id = 'file';
        __file.accept = '.jpg';
        __file.required = true;
        p.appendChild( __file);
        form.appendChild(p);
        p = document.createElement('p');
        txt = document.createTextNode('Описание');
        p.appendChild(txt);
        br = document.createElement('br');
        p.appendChild(br);
        __description = document.createElement('textarea');
        __description.id = 'description';
        p.appendChild(__description);
        form.appendChild(p);
        p = document.createElement('p');
        let input = document.createElement('input');
        input.type = 'submit';
        input.value = 'Добавить';
        p.appendChild(input);
        form.appendChild(p);
        document.body.appendChild(form);
        form.addEventListener('submit', sendData);
        p = document.createElement('p');
        let a = document.createElement('a');
        a.href = '';
        a.textContent = 'Назад';
        p.appendChild(a);
        document.body.appendChild(p);
        document.title = 'Добавление изображения';
        window.scrollTo(0, 0);
    }
    return PGAdd;
})();
```

Конструктор класса выводит на экран заголовок, веб-форму для выбора изображения и гиперссылку **Назад**. К форме привязывается обработчик события `submit` — функция `sendData`.

Переменные `__file` и `__description`, хранящие, соответственно, поле для выбора файла с изображением и область редактирования для ввода описания, должны быть доступны не только в конструкторе, но и в функции `sendData`. Поэтому мы объявили их непосредственно в замыкании, как и переменную `__ajax`, в которую будет занесен объект AJAX.

Функция `sendData` создает объект AJAX, с его помощью отправляет веб-службе add.php изображение и описание к нему (если оно введено). Функция `redirect`, вызывающаяся после получения результата от веб-службы, выполняет переход по «пустому» интернет-адресу, что вызовет перенаправление на список миниатюр.

5. Откроем страницу index.html в текстовом редакторе и дополним ее код следующим образом:

```
<html>
    <head>
        . . .
        <script src="pgadd.js" type="text/javascript"></script>
    </head>
    <body>
    </body>
</html>
<script type="text/javascript">
    . . .
    function route() {
        . . .
        if (result = getRE.exec(hash))
            component = new PGItem(result[1]);
        else if (hash == '#add')
            component = new PGAdd();
        else if (!hash)
            component = new PGList();
    }
    . . .
</script>
```

Запустим веб-сервер и откроем наш сайт. Перейдем на компонент добавления изображения и загрузим в галерею новую картинку.

19.6. Самостоятельные упражнения

- Сделайте так, чтобы при переходе от просмотра выбранного изображения на список миниатюр этот список прокручивался к изображению, просматривавшемуся ранее.
- Реализуйте удаление изображений.
- Сделайте так, чтобы при указании в интернет-адресе якоря неподдерживаемого формата выводилось предупреждение о том, что такая страница отсутствует («ошибка 404»), и гиперссылка на список миниатюр.

Урок 20
Серверные сообщения

Серверные сообщения
Поддержание соединения
Объявление своих событий

20.1. Использование серверных сообщений

Если веб-обозревателю понадобится какой-либо файл, хранящийся на веб-сервере (страница, изображение, внешняя таблица стилей, JSON-данные, загружаемые посредством AJAX), он отправит клиентский запрос на загрузку этого файла.

Если же веб-службе понадобится переслать что-либо веб-обозревателю, у нее два выхода из этого положения:

- ждать, когда веб-обозреватель проявит инициативу и соизволит отправить запрос к ней;
- использовать технологию серверных сообщений.

> *Серверное сообщение* — текстовое сообщение, пересылаемое веб-службой веб-обозревателю по своей инициативе (а не по инициативе веб-обозревателя).

20.1.1. Что представляет собой серверное сообщение?

Серверное сообщение:

- формируется в виде обычной строки;
- записывается в одну строку в формате:

  `data: <`*`содержание сообщения`*`>.`

  Пример:

  ```
  data: Это серверное сообщение. Всем привет!
  ```

  Пробел между двоеточием в префиксе `data:` и *`содержанием сообщения`* необязателен;

- может иметь в качестве *`содержания`* как обычный текст, так и данные в формате JSON:

  ```
  data: {"type": "greeting", "content": "Всем привет!"}
  ```

- отделяется от следующего серверного сообщения двойным разрывом строки:

```
data: (Сообщение 1) Это ваша веб-служба.

data: (Сообщение 2) Здравствуйте!

data: (Сообщение 3) Не соскучились?
```

- помимо *содержания*, может включать служебные данные, помечаемые другими префиксами и отправляемые вслед за содержанием. Эти служебные данные мы рассмотрим позже.

20.1.2. Отправка серверных сообщений

Для отправки серверных сообщений бэкенд должен:

- указать в заголовке `Content-Type` в качестве MIME-типа пересылаемых данных `text/event-stream` (для добавления заголовка в PHP можно использовать функцию `header`);
- отключить кеширование сообщений в веб-обозревателе, задав заголовок `Cache-Control` со значением `no-cache`;
- сформировать текст очередного сообщения, завершив его двумя разрывами строк — двумя строковыми литералами `\n\n`;
- отправить сообщение оператором вывода `echo`;
- инициировать принудительную отправку сообщения веб-обозревателю, вызвав не принимающую параметров функцию `flush` (если этого не сделать, сообщения будут отправлены только после завершения работы веб-службы).

Вот пример веб-службы ssm.php, отправляющей два серверных сообщения:

```
<?php
header('Content-Type: text/event-stream');
header('Cache-Control: no-cache');
echo "data: Здравствуйте! Это ваша веб-служба.\n\n";
flush();
echo "data: Не соскучились?\n\n";
flush();
?>
```

20.1.3. Прием серверных сообщений

Для получения серверных сообщений фронтенд должен:

- создать оператором `new` объект класса `EventSource`:

```
new EventSource(<интернет-адрес веб-службы, ↵
отправляющей серверные сообщения>)
```

Сразу после этого веб-обозреватель установит соединение с веб-службой, тем самым запустив ее на выполнение, и станет ожидать поступления сообщений от нее;

- привязать к событию `message` этого объекта, возникающему при получении серверного сообщения, обработчик;
- в теле этого обработчика получить содержание сообщения из свойства `data` объекта события. Если содержание сообщения представляет собой JSON-данные, — декодировать их (см. *разд. 17.4*).

Вот пример соединения с веб-службой ssm.php и обработки отправляемых ей серверных сообщений:

```
const es = new EventSource('ssm.php');
es.addEventListener('message', (evt) => {
    let messageData = evt.data;
    // Обрабатываем полученное сообщение
};
```

Класс `EventSource` поддерживает свойство `readyState`, хранящее числовое обозначение текущего состояния соединения:

- `0` — соединение с веб-службой устанавливается;
- `1` — соединение установлено, идет прием сообщений;
- `2` — соединение закрыто.

Помимо события `message`, класс `EventSource` поддерживает следующие события:

- `open` — возникает сразу после установки соединения с веб-службой;
- `error` — возникает при разрыве соединения в результате сетевых неполадок.

20.1.4. Поддержание соединения и возобновление приема

При внезапном разрыве соединения в результате сетевых неполадок веб-обозреватель будет выполнять попытки восстановить его, пока они не увенчаются успехом.

Продолжительность паузы между отдельными попытками восстановить соединение у разных веб-обозревателей различается и может составлять 1–3 сек. Веб-служба может явно указать продолжительность паузы, отправив в сообщении строку формата:

```
retry: <продолжительность паузы в миллисекундах>
```

Пример:

```
retry: 60000
data: Нам, веб-службам, торопиться некуда...
```

Для того чтобы веб-обозреватель смог возобновить получение сообщений с момента его разрыва, нужно, чтобы веб-служба пересылала в составе каждого сообщения — *обязательно в самом его конце* — строку следующего формата:

```
id: <уникальный идентификатор сообщения>
```

Пример:

```
data: Это ваша веб-служба.
id: 1

data: Здравствуйте!
id: 2

data: Не соскучились?
id: 3
```

Получив такой *идентификатор*, веб-обозреватель занесет его в свойство `lastEventId` объекта класса `EventSource` и после возобновления соединения отправит его серверу в заголовке `Last-Event-ID` клиентского запроса. Получив такой заголовок, веб-служба должна выслать все сообщения, следующие за сообщением с полученным идентификатором.

20.1.5. Создание своих событий

Есть возможность создать свои события, которые будут возникать в объекте класса `EventSource` и могут быть обработаны наравне со встроенными.

Веб-служба должна отправить в составе сообщения — *в самом его начале* — строку формата:

```
event: <имя создаваемого события>
```

Имя события должно удовлетворять тем же требованиям, что предъявляются к именам переменных JavaScript (см. *разд. 1.7*).

Пример:

```
event: newfile
data: {"id": "20190402114732", "name": "20190402114732.jpg"}
```

На стороне фронтенда следует привязать к этому событию обработчик:

```
es.addEventListener('newfile', (evt) => {
    const data = JSON.parse(evt.data);
    let fileID = data.id;
    let fileName = data.name;
    // Что-либо делаем с полученными данными
};
```

20.1.6. Разрыв соединения

Веб-обозреватель будет постоянно поддерживать соединение с веб-службой, даже если та уже завершила свою работу. Чтобы явно разорвать соединение, нужно вызвать не принимающий параметров метод `close` объекта класса `EventSource`:

```
ex.close();
```

20.2. Упражнение. Новая фотогалерея. Более отзывчивый список миниатюр

В фотогалерее, написанной в *уроке 19*, веб-служба get.php формирует перечень миниатюр в виде массива. Если изображений в фотогалерее много, этот массив окажется очень большим, и фронтенду потребуется много времени, чтобы загрузить его и вывести на экран. Сделаем так, чтобы бэкенд пересылал этот перечень фронтенду в серверных сообщениях небольшими порциями — по одной миниатюре каждая.

1. Скопируем в корневую папку веб-сервера из состава пакета XAMPP все содержимое папки 20\!sources из сопровождающего книгу электронного архива (см. *приложение 3*).
2. Для пересылки перечня миниатюр в серверных сообщениях мы напишем веб-службу get2.php (а веб-службу get.php используем, как и ранее, для получения сведений о выбранном изображении).

   Создадим в корневой папке веб-сервера файл get2.php со следующим кодом:

```
<?php
header('Content-Type: text/event-stream');
header('Cache-Control: no-cache');
$images_folder = '/images/';
$current_dir = dirname($_SERVER['SCRIPT_FILENAME']);
$path = $current_dir . $images_folder;
$last_id = (isset($_SERVER['LAST_EVENT_ID'])) ?
    $_SERVER['LAST_EVENT_ID'] : 0;
$n = 1;
if ($dh = opendir($path)) {
    while (($file = readdir($dh)) !== false){
        if ($file != '.' && $file != '..') {
            if ($n > $last_id) {
                $el['path'] = $images_folder . $file;
                $el['id'] = pathinfo($file, PATHINFO_FILENAME);
                echo "event: newfile\n";
                echo 'data: ' . json_encode($el) . "\n";
                echo 'id: ' . $n . "\n\n";
                flush();
                $n++;
            }
        }
    }
    closedir($dh);
}
echo "event: end\n";
echo "data: \n\n"
?>
```

Идентификатор последнего полученного фронтендом серверного сообщения, отправленный в заголовке `Last-Event-ID`, хранится в ассоциативном массиве `$_SERVER` в элементе с ключом `LAST_EVENT_ID`. Он понадобится нам, чтобы возобновить передачу миниатюр в случае разрыва соединения.

Мы создали два своих события:

- `newfile` — для передачи сведений об очередной миниатюре. Сведения передаются в виде JSON-нотации объекта со свойствами `id` и `path`;
- `end` — для завершения передачи. Передается вместе с «пустым» содержанием.

И не забываем, что в PHP строковые литералы обрабатываются только в строках, взятых в двойные кавычки.

3. Для вывода нового списка миниатюр напишем компонент `PGList2` — создадим в корневой папке файл pglist2.js с кодом этого компонента:

```
const PGList2 = (function () {
    let __section;
    function newFile(evt) {
        const el = JSON.parse(evt.data);
        const a = document.createElement('a');
        a.href = '#get=' + el.id;
        const img = document.createElement('img');
        img.src = el.path;
        a.appendChild(img);
        __section.appendChild(a);
    }
    function end(evt) {
        evt.target.close();
    }
    function PGList2() {
        document.body.innerHTML = '';
        let h1 = document.createElement('h1');
        h1.textContent = 'Фотогалерея';
        document.body.appendChild(h1);
        let p = document.createElement('p');
        let a = document.createElement('a');
        a.href = '#add';
        a.textContent = 'Добавить изображение';
        p.appendChild(a);
        document.body.appendChild(p);
        __section = document.createElement('section');
        __section.id = 'photogallery';
        document.body.appendChild(__section);
        document.title = 'Фотогалерея';
        window.scrollTo(0, 0);
        this.es = new EventSource('get2.php');
        this.es.addEventListener('newfile', newFile);
```

```
            this.es.addEventListener('end', end);
        }
        return PGList2;
    })();
```

В конструкторе на страницу выводятся заголовок, гиперссылка на компонент для загрузки изображения и «пустой» тег `<section>`, в котором будут выводиться миниатюры. После этого устанавливается соединение с веб-службой get2.php и выполняется привязка обработчиков к событиям `newfile` и `end`.

Обработчик события `newfile` создаст гиперссылку с миниатюрой и выведет ее в созданный ранее тег `<section>`. Обработчик события `end` просто закроет соединение с веб-службой.

4. Откроем файл страницы index.html в текстовом редакторе и исправим код таким образом, чтобы он использовал компонент `PGList2` (исправления выделены полужирным шрифтом):

```
<!doctype html>
<html>
    <head>
        . . .
        <script src="pglist2.js" type="text/javascript"></script>
        . . .
    </head>
</html>
<script type="text/javascript">
    . . .
    function route() {
        . . .
        else if (!hash)
            component = new PGList2();
    }
    . . .
</script>
```

Запустим веб-сервер, откроем нашу фотогалерею и убедимся, что список миниатюр успешно выводится.

Заключение

На этом книга, посвященная языку программирования JavaScript, закончена. Мы изучили все наиболее востребованные инструменты самого языка, классов DOM, BOM, HTML API и даже немного познакомились с платформой серверных приложений PHP. Мы запомнили множество новых терминов, языковых конструкций, классов, свойств и методов. Мы написали десятки учебных веб-страниц, в том числе пару вполне функциональных интернет-фотогалерей. И теперь с гордостью можем называть себя веб-программистами.

За рамками рассмотренного в книге осталось не так уж и много:

- специфические, редко используемые инструменты: класс сведений о веб-обозревателе `Navigator`, класс истории веб-обозревателя `History`, средства геолокации, программного создания окон и др.;
- инструменты, недавно введенные в стандарт языка, пока существующие в черновых редакциях и поддерживаемые не всеми веб-обозревателями: HTML-шаблоны, теневая DOM, «настоящие» компоненты и т. п.;
- устаревшие инструменты, оставленные для совместимости и не рекомендуемые к использованию.

Автор полагает, что если читатель этим заинтересуется, то без особых проблем сможет найти нужную информацию в Сети. Благо он, после прочтения этой книги, уже имеет знания, необходимые для знакомства с упомянутыми инструментами.

Книга, которую вы, уважаемые читатели, только что закончили, описывает программирование на JavaScript только клиентских веб-сценариев. На самом деле в настоящее время этот замечательный язык применяется гораздо шире, в том числе для разработки серверных веб-приложений, бэкендов и обычных оконных программ. Это стало возможным после появления программной платформы Node.js, исполняющей JavaScript-код вне веб-обозревателя.

На языке JavaScript написан ряд библиотек, упрощающих выполнение типичных задач веб-программирования, наподобие привязки обработчиков к событиям (jQuery, Prototype, rxjs и многие другие), сложные библиотеки (*фреймворки*) для написания фронтендов (Angular, React, Vue) и бэкендов (Express, Meteor и др.).

Существуют даже производные языки, призванные устранить некоторые недостатки JavaScript (в частности, неудобный синтаксис объявления классов): TypeScript, CoffeeScript и Dart. Программы, написанные на этих языках, преобразуются в обычный JavaScript-код, исполняемый веб-обозревателем (или платформой Node.js).

Так что у JavaScript — бурное настоящее и многообещающее будущее. И изучать его, безусловно, стоит!

А чтобы изучение проходило эффективно и увлекательно — воспользуйтесь следующими полезными интернет-ресурсами по теме книги:

- **https://developer.mozilla.org/ru/docs/Web/JavaScript** — исчерпывающее руководство и справочник по JavaScript, включающий вновь введенные в стандарт инструменты, пока еще не поддерживаемые всеми веб-обозревателями. Часть материалов написана на английском языке;
- **https://www.w3schools.com/** — крупный англоязычный сайт, посвященный всевозможным интернет-технологиям: HTML, CSS, JavaScript, PHP и пр.;
- **https://vk.com/we_use_js** — сообщество JavaScript-программистов в социальной сети «ВКонтакте»;
- **https://vk.com/nisnom** — другое сообщество социальной сети «ВКонтакте», посвященное, помимо того, HTML, CSS и PHP.

На этом автор прощается с вами, уважаемые читатели! До свидания и успехов на ниве JavaScript-программирования!

Владимир Дронов

Приложение 1
Приоритет операторов

Приоритет	Оператор	Описание	Пример
13	(. . .)	Круглые скобки	(10 + 2) / 3
12	. (точка)	Доступ к элементу объекта	arr.length
	[]	Доступ к элементу массива	arr[1]
	()	Вызов функции	func(1, 2)
	new	Создание объекта	new Date()
11	++	Инкремент, поставленный после переменной	i++
	--	Декремент, поставленный после переменной	count--
10	++	Инкремент, поставленный перед переменной	++i
	--	Декремент, поставленный перед переменной	--count
	!	Логическое НЕ	!flag
	typeof	Получение типа	typeof s
9	**	Возведение в степень	5 ** 3
8	*	Умножение	4 * 3
	/	Деление	10 / 3
	%	Остаток от деления	10 % 3
7	+	Сложение	1 + 2
	-	Вычитание	2 - 1
6	<	Меньше	x < y
	<=	Меньше или равно	x <= y
	>	Больше	x > y
	>=	Больше или равно	x >= y
	in	Проверка существования свойства	'prop' in obj
	instanceof	Принадлежность классу	d instanceof Date

(окончание)

Приоритет	Оператор	Описание	Пример
5	==	Равно	x == y
	===	Строго равно	x === y
	!=	Не равно	x != y
	!==	Строго не равно	x !== y
4	&&	Логическое И	flag1 && flag2
3	||	Логическое ИЛИ	flag1 || flag2
2	? . . . :	Условный	(flag) ? x : y
1	=, +=, -=, *=, /=, %=	Присваивание	x = y

Приложение 2
Пакет хостинга XAMPP

Установка XAMPP
Запуск и остановка комплектного веб-сервера
Использование комплектного веб-сервера
Решение проблем

П2.1. Установка пакета хостинга XAMPP

Для тестирования страниц, разрабатываемых в *уроках 17–20*, нам понадобится работающий веб-сервер и платформа PHP. И то, и другое входит в пакет хостинга XAMPP, который мы сейчас установим.

> XAMPP — программный пакет интернет-хостинга, включающий в свой состав все необходимое для запуска веб-сайта: веб-сервер Apache HTTP Server, программную платформу PHP, серверную СУБД MariaDB и ряд других программ. Все эти программы полностью работоспособны и не требуют ни специального сопряжения, ни дополнительной настройки.

1. Перейдем в веб-обозревателе по интернет-адресу **https://www.apachefriends.org/ru/index.html**.
2. На открывшейся странице найдем большую гиперссылку **XAMPP для Windows** и щелкнем на ней (рис. П2.1).
3. Сохраним загруженный файл с дистрибутивом XAMPP где-либо на локальном диске. Запустим его, положительно ответим на запрос подсистемы контроля доступа UAC (текст в этом окне-сообщении предостерегает от установки XAMPP в папку C:\Program Files\ или в любую другую папку, защищенную UAC) и закроем это окно-сообщение, нажав кнопку **ОК**.
4. Нажмем в открывшемся диалоговом окне **Setup XAMPP** кнопку **Next** (рис. П2.2).
5. В следующем диалоговом окне **Select Components** (рис. П2.3) выберем Apache и PHP, установив в иерархическом списке компонентов флажки **Server | Apache** и **Program Languages | PHP**, и нажмем кнопку **Next**.

   Изменим при необходимости путь установки (по умолчанию — C:\xampp), введя его в поле **Select a folder** следующего диалогового окна **Installation folder** (рис. П2.4), и нажмем кнопку **Next**. Вы также можете щелкнуть на расположенной правее этого поля кнопке и выбрать в открывшемся диалоговом окне другую папку для установки. Впрочем, путь, предлагаемый по умолчанию, подойдет в большинстве случаев.

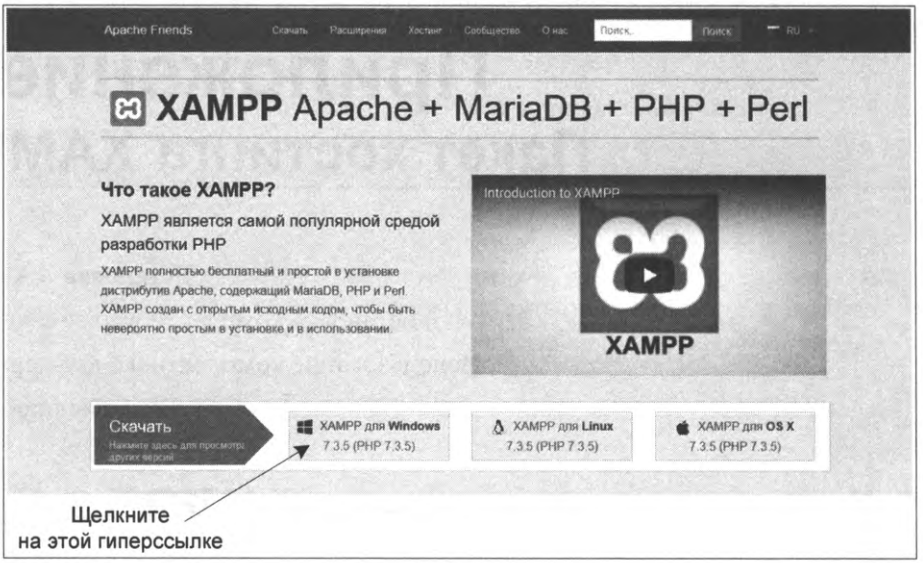


Рис. П2.1. Гиперссылка **XAMPP для Windows**

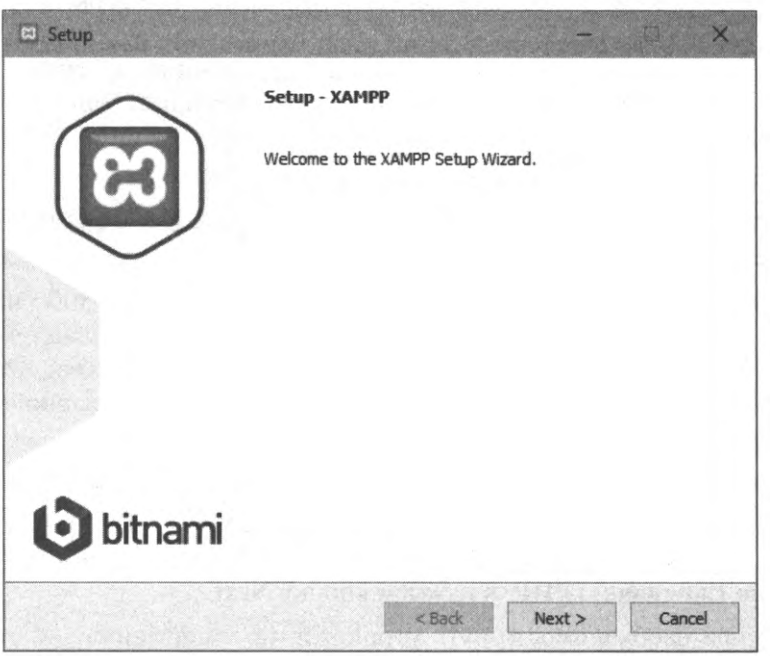


Рис. П2.2. Стартовое диалоговое окно **Setup XAMPP**

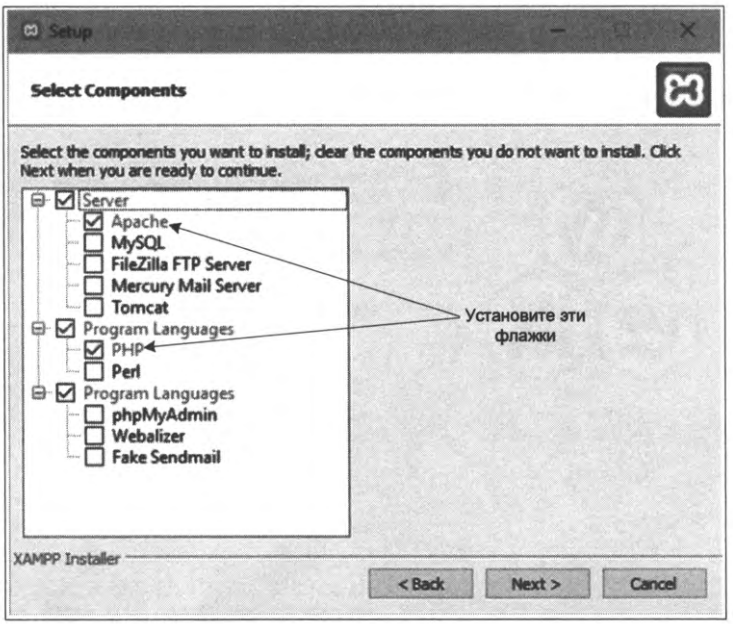


Рис. П2.3. Диалоговое окно **Select Components**

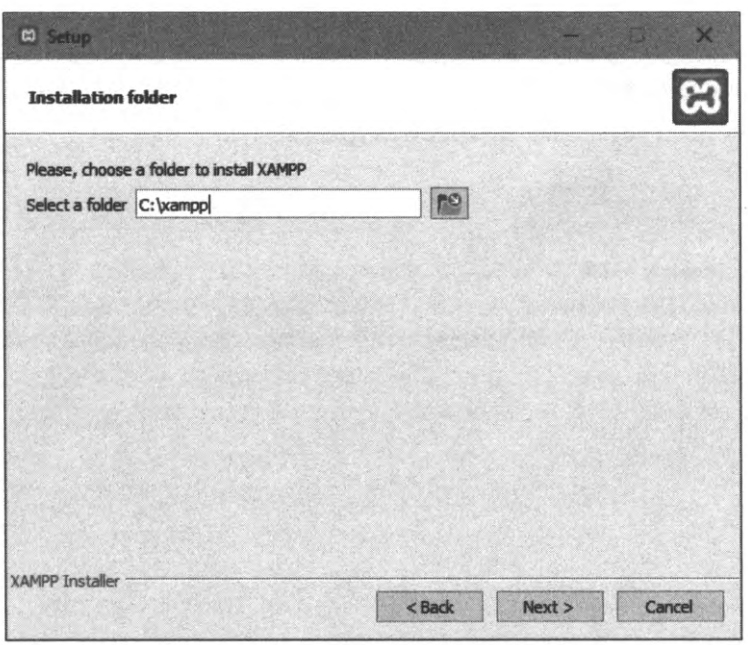


Рис. П2.4. Диалоговое окно **Installation folder**

6. Сбросим в следующем диалоговом окне **Bitnami for XAMPP** (рис. П2.5) флажок **Learn more about Bitnami for XAMPP**, чтобы установщик не вывел веб-страницу с дополнительными сведениями (сейчас они нам не нужны), и нажмем кнопку **Next**.

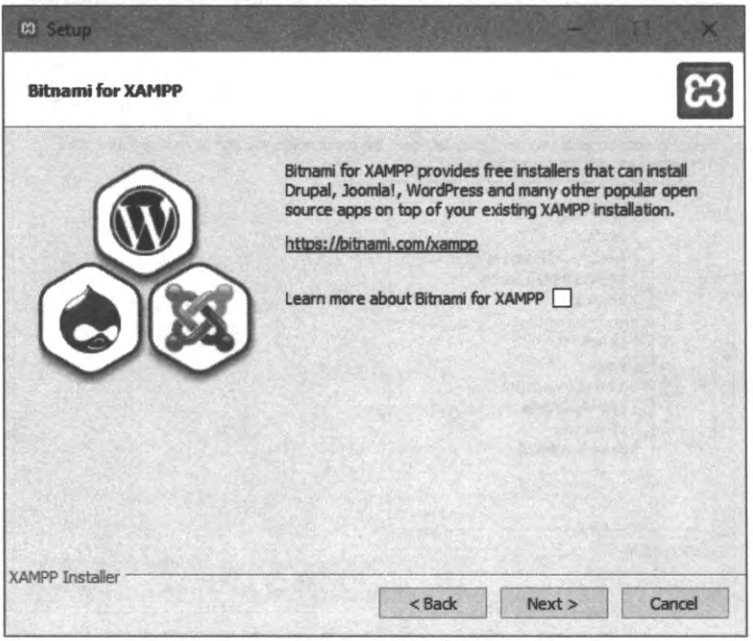


Рис. П2.5. Диалоговое окно **Bitnami for XAMPP**

7. В следующем диалоговом окне **Ready to Install** (рис. П2.6) нажмем кнопку **Next**, чтобы непосредственно начать установку пакета.
8. Подождем, пока установка завершится (рис. П2.7).

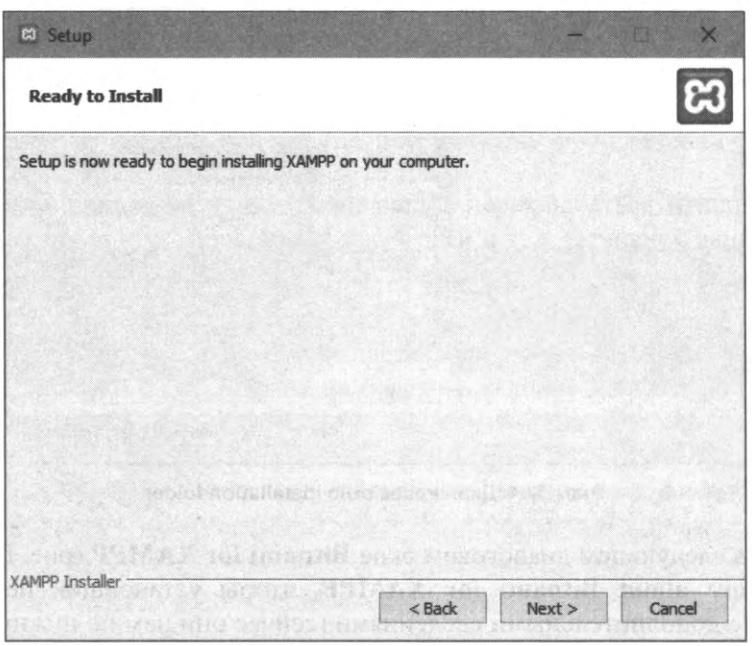


Рис. П2.6. Диалоговое окно **Ready to Install**

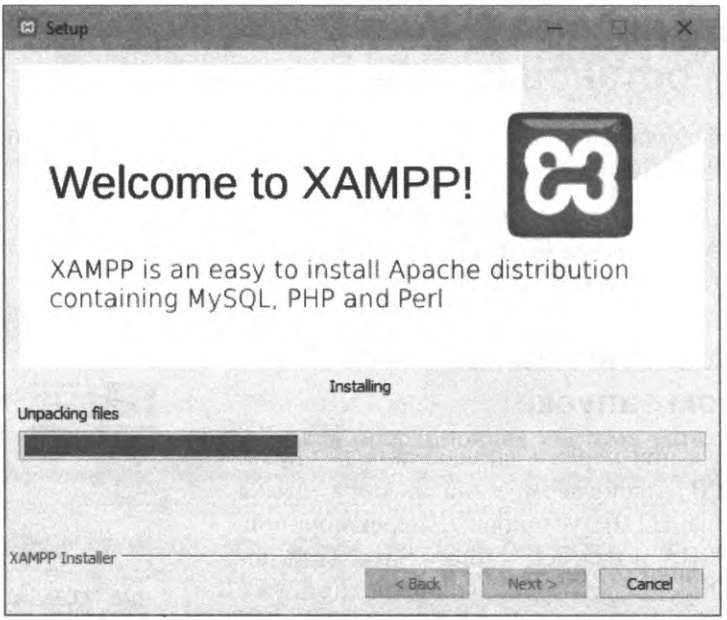


Рис. П2.7. Окно, отображающее ход процесса установки пакета

В завершающем установку диалоговом окне **Completing the XAMPP Setup Wizard** (рис. П2.8) нажмем кнопку **Finish**, чтобы закрыть установщик. Если флажок **Do you want to start the Control Panel now?** не был сброшен, запустится панель управления XAMPP (см. далее).

Рис. П2.8. Диалоговое окно **Completing the XAMPP Setup Wizard**

П2.2. Панель управления XAMPP. Запуск и остановка веб-сервера

> *Панель управления XAMPP* — утилита для управления компонентами пакета XAMPP: запуска, остановки и настройки.

Открывается панель управления XAMPP непосредственным запуском исполняемого файла xampp-control.exe, находящегося в папке, где установлен пакет. К сожалению, запустить ее из меню **Пуск** невозможно.

П2.2.1. Указание языка при первом запуске

В открывшемся при первом запуске панели управления XAMPP диалоговом окне выбора языка **Language** (рис. П2.9) установим переключатель под флагом США и нажмем кнопку **Save**. При последующих запусках эту процедуру выполнять уже не придется.

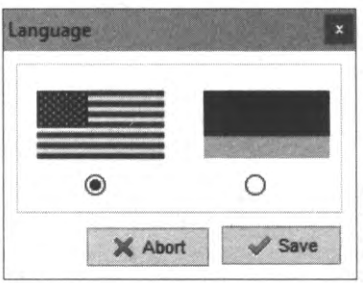


Рис. П2.9. Диалоговое окно выбора языка **Language**

П2.2.2. Окно панели управления XAMPP

Окно панели управления XAMPP по горизонтали делится на две части (рис. П2.10):

- *вверху* — таблица с перечнем всех установленных компонентов и наборы кнопок для управления ими (запуска, остановки и др.);

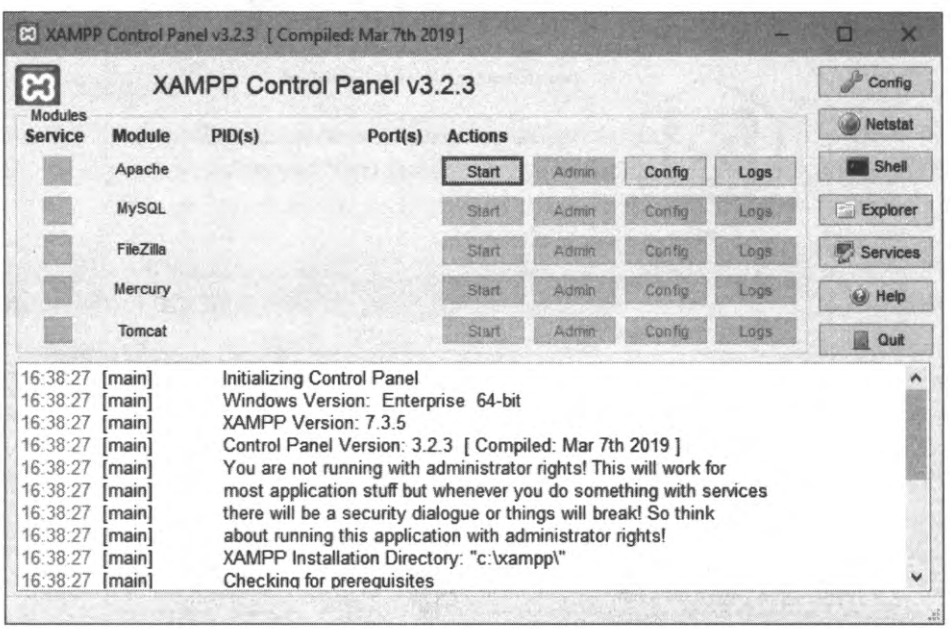


Рис. П2.10. Окно панели управления XAMPP

- *внизу* — журнал работы пакета, в котором сохраняются основные сообщения, выводимые его компонентами.

П2.2.3. Запуск веб-сервера

Найдем в строке **Apache** (самой верхней) перечня компонентов панели управления XAMPP кнопку **Start** (рис. П2.11) и нажмем ее. После этого надпись на кнопке сменится на **Stop**.

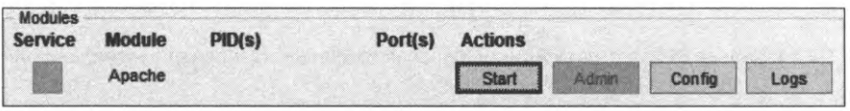


Рис. П2.11. Кнопка **Start** в строке **Apache** перечня компонентов в панели управления XAMPP

П2.2.4. Проверка работоспособности веб-сервера

В веб-обозревателе перейдем по интернет-адресу **http://localhost/** — на экране появится начальная страница небольшого тестового сайта, поставляемого в составе XAMPP (рис. П2.12).

Рис. П2.12. Начальная страница тестового сайта, поставляемого в составе XAMPP

П2.2.5. Остановка веб-сервера

Найдем в строке **Apache** (самой верхней) перечня компонентов панели управления XAMPP кнопку **Stop** (рис. П2.13) и нажмем ее. После этого надпись на кнопке сменится на **Start**.

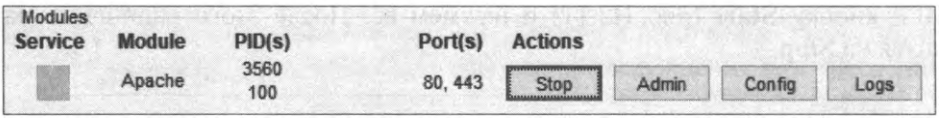


Рис. П2.13. Кнопка **Stop** в строке **Apache** перечня компонентов в панели управления XAMPP

П2.3. Использование веб-сервера

П2.3.1. Тестирование веб-сайта с применением XAMPP

1. Откроем папку по пути *<путь установки XAMPP>*\htdocs. Это корневая папка веб-сервера Apache HTTP Server из комплекта XAMPP.

   Изначально там хранится тестовый сайт, позволяющий проверить работоспособность пакета. Рекомендуется сохранить этот сайт на всякой случай.

2. Если в корневой папке хранятся файлы тестового сайта, переименуем ее, например, в _htdocs, и создадим в папке *<путь установки XAMPP>* новую корневую папку htdocs.

3. Скопируем файлы сайта, который хотим протестировать, в папку *<путь установки XAMPP>*\htdocs.

Не забываем запускать веб-сервер перед тестированием сайта и останавливать перед крупными правками.

П2.3.2. Просмотр журналов работы веб-сервера и PHP

> *Журнал* (*лог*) — файл для сохранения сведений о работе программы-сервера: выполненных ею действиях, нештатных ситуациях и пр. Практически всегда имеет текстовый формат и расширение log.

1. Найдем в строке **Apache** (самой верхней) перечня компонентов панели управления XAMPP кнопку **Logs** (см. рис. П2.11) и нажмем ее.

2. Выберем в появившемся на экране меню (рис. П2.14) пункт:

   - **Apache (access.log)** — для просмотра журнала веб-сервера, содержащего сведения об отправленных им файлах;
   - **Apache (error.log)** — для просмотра журнала ошибок, возникших в работе веб-сервера (в частности, отсутствующих файлах). Также включает сообщения об ошибках в PHP-коде.

   Соответствующий журнал будет открыт в текстовом редакторе Блокнот.

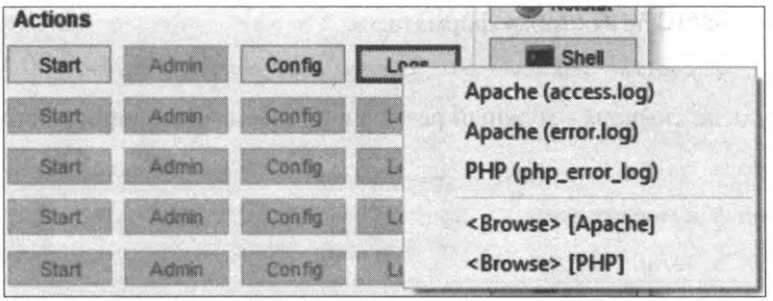


Рис. П2.14. Меню журналов работы веб-сервера и PHP

Примечание

В меню также присутствует пункт **PHP (php_error_log)**, но при его выборе выводится сообщение о том, что файл с таким журналом отсутствует. Судя по всему, это программная ошибка в пакете XAMPP.

П2.4. Решение проблем

П2.4.1. Увеличение максимального размера выгружаемых файлов

По умолчанию PHP позволяет загружать на сервер файлы объемом не более 2 Мбайт (если загружаются сразу несколько файлов, учитывается их совокупный объем). Чтобы избавиться от этого ограничения:

1. Запустим панель управления XAMPP.
2. Остановим веб-сервер, если он запущен.
3. Найдем в строке **Apache** (самой верхней) перечня компонентов кнопку **Config** (см. рис. П2.11) и нажмем ее.
4. Выберем в появившемся на экране меню (рис. П2.15) пункт **PHP (php.ini)** — конфигурационный файл php.ini, хранящий настройки PHP, будет открыт в Блокноте.

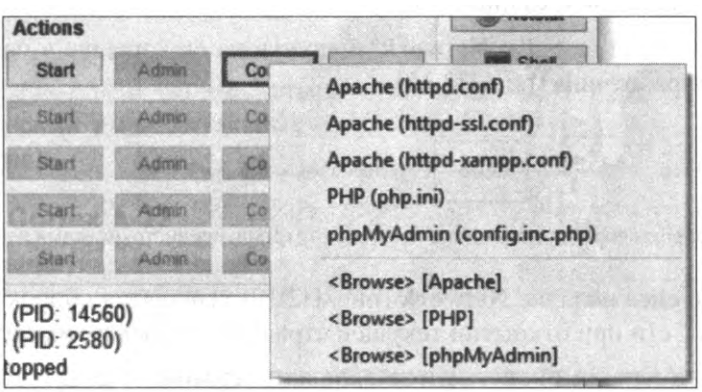


Рис. П2.15. Меню кнопки **Config** веб-сервера **Apache** — выбираем пункт **PHP (php.ini)**

5. Найдем в файле php.ini строку формата:

   `upload_max_filesize=<максимальный размер выгружаемого файла>`

6. Укажем после символа = нужный размер в одном из следующих форматов:

   - `<размер в байтах>`
   - `<размер в килобайтах>K`
   - `<размер в мегабайтах>M`
   - `<размер в гигабайтах>G`

7. Сохраним и закроем файл php.ini.

После этого можно запустить веб-сервер и проверить, как работает загрузка файлов на сервер.

П2.4.2. Отключение кеширования файлов веб-обозревателем

В процессе разработки сайта может возникнуть ситуация, когда изменив в корневой папке веб-сервера какой-либо файл и перезагрузив открытую в веб-обозревателе страницу, мы увидим, что веб-обозреватель все еще использует устаревшую копию файла, взятую из кеша.

> *Кеш* — хранилище, находящееся на локальном диске и используемое веб-обозревателем для сохранения копий всех загруженных им файлов для ускорения последующего доступа к ним.

По идее, веб-обозреватель должен хранить в кеше наиболее актуальные версии файлов, загружая их повторно, если они изменятся. Но иногда устаревшие копии файлов *«застревают»* в кеше и не перезаписываются более новыми.

Проще всего решить проблему «застревания» файлов в кеше, отключив кеширование файлов веб-обозревателем.

1. Выведем панель с инструментами разработчика, встроенными в веб-обозреватель, нажав клавишу <F12>.
2. Переключимся на вкладку **Network**, щелкнув на ее корешке в панели с инструментами разработчика (рис. П2.16).

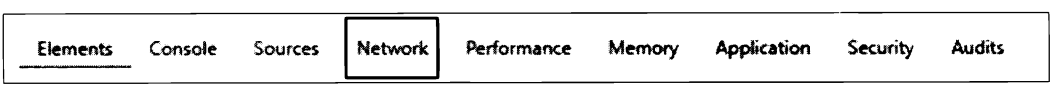


Рис. П2.16. Корешок вкладки **Network** на панели с отладочными инструментами веб-обозревателя

На открывшейся вкладке **Network** (рис. П2.17) выводятся список файлов, загруженных по Сети при открытии текущей страницы, и график их загрузки.

3. Установим в панели инструментов вкладки **Network**, находящейся над графиком, флажок **Disable cache** (рис. П2.18).

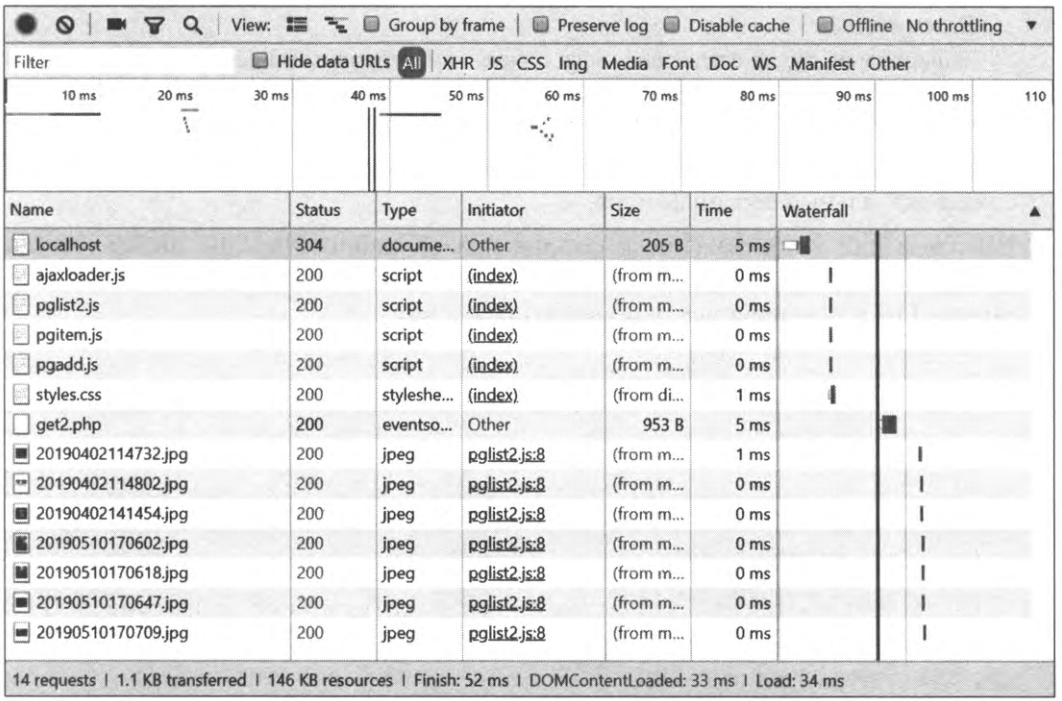


Рис. П2.17. Вкладка **Network** панели с инструментами разработчика

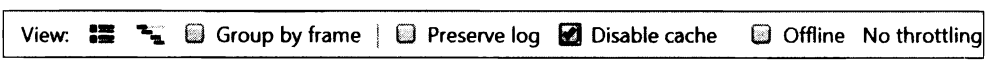


Рис. П2.18. Панель инструментов вкладки **Network** — устанавливаем флажок **Disable cache**

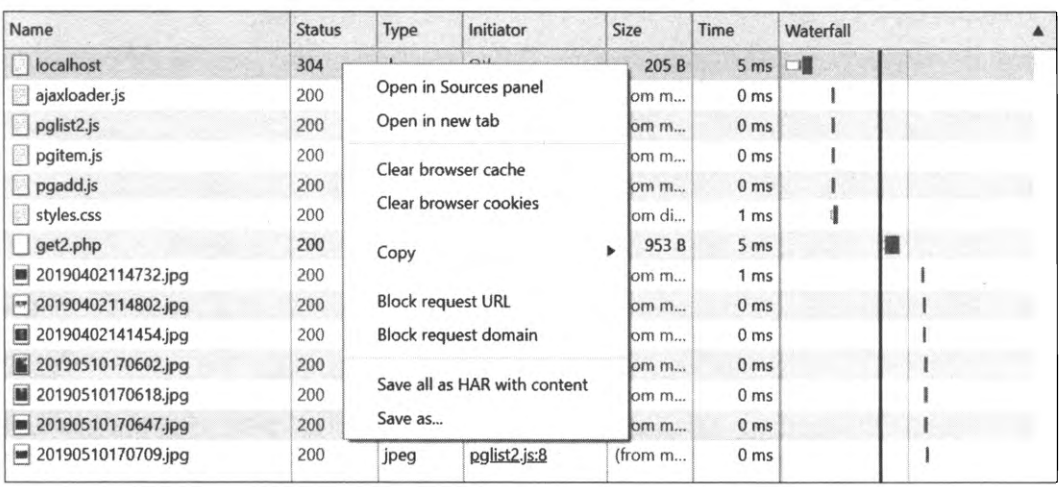


Рис. П2.19. Контекстное меню списка файлов вкладки **Network** — выбираем пункт **Clear browser cache**

Внимание!

Кеширование будет отключено, пока панель с инструментами разработчика присутствует на экране. Если закрыть ее, кеширование вновь активизируется, независимо от состояния флажка **Disable cache**.

4. Напоследок необходимо очистить кеш веб-обозревателя, чтобы тот использовал самые актуальные версии файлов.

   Щелкнем для этого на списке файлов правой кнопкой мыши, выберем в появившемся на экране меню пункт **Clear browser cache** (рис. П2.19) и нажмем кнопку **ОК** в появившемся окне-предупреждении.

Приложение 3
Описание электронного архива

Электронный архив, сопровождающий книгу, размещен на FTP-сервере издательства «БХВ-Петербург» по интернет-адресу: **ftp://ftp.bhv.ru/9785977565899.zip**. Ссылка на него доступна и со страницы книги на сайте **http://www.bhv.ru/**.

Список папок и файлов, имеющихся в архиве (вложенность папок и файлов показана отступами):

- *<номер урока>* — папка с материалами урока с указанным *номером*. Состав папки:
  - !sources — папка с исходными материалами, необходимыми для выполнения упражнений в текущем уроке;
  - ex*<номер раздела>* — результаты выполнения упражнения, приведенного в разделе с указанным *номером*;
  - s*<номер раздела>* — результаты выполнения самостоятельных упражнений, приведенных в разделе с указанным *номером*;
- readme.txt — текстовый файл с описанием электронного архива.

Предметный указатель

$

$_FILES 296
$_GET 293
$_POST 296
$_SERVER 290

A

abort 151, 209
abs 80
acos 81
acosh 81
activeElement 124
add 136, 156
addColorStop 233
addEventListener 106, 116
afterprint 110, 187
AJAX 271
alert 192
altKey 114, 115
animationend 110, 115
AnimationEvent 115
animationiteration 110, 115
animationName 115
animationstart 110, 115
append 310
appendChild 142
apply 254
arc 230
arcTo 230
Arguments 82
Array 71
asin 81
asinh 81
assign 191
atan 81
atan2 81
atanh 81
autocomplete 155, 164
availHeight 192
availWidth 192

B

beforeprint 110, 187
beforeunload 110, 187
beginPath 227
bezierCurveTo 232
blur 156, 157, 186, 187
body 123
BOM 185
Boolean 67
bottom 130
break 42, 47
bubbles 118
button 114

C

call 256
cancellable 121
canplay 150
canplaythrough 150
CanvasGradient 233
CanvasPattern 236
CanvasRenderingContext2D 221
cbrt 80
ceil 80
change 157
characterSet 144
charAt 70
charCode 114
charCodeAt 71
checked 156
checkValidity 166
childElementCount 126
children 126
classList 136
className 136
clear 212
clearData 214
clearInterval 200
clearRect 222
clearTimeout 202
click 109

clientHeight 128
clientWidth 128
clientX 114
clientY 114
clip 240
ClipboardEvent 115
close 320
closedir 292
closePath 228
colorDepth 192
complete 147
concat 76
confirm 192
Console 89, 90
const 27, 41
constructor 254
contains 126, 137
contextmenu 109
continue 47
copy 109
cos 81
cosh 81
create 254
createElement 141
createLinearGradient 233
createPattern 236
createRadialGradient 235
createTextNode 141
CSSStyleDeclaration 137
ctrlKey 114, 115
currentSrc 149
currentTarget 113
currentTime 148
customError 168
cut 157

D

data 319
Data URL 210
dataTransfer 213
DataTransfer 213
date 287
Date 78
dblclick 109
defaultChecked 156
defaultMuted 149
defaultPlaybackRate 149
defaultPrevented 121
defaultSelected 156
defaultValue 155
defineProperty 251, 252
delete 85
deltaMode 114
deltaX 114
deltaY 114
dirname 290
document 123
DOM 104
domain 144
DOMRect 130
DOMTokenList 136
drag 213
Drag'n'drop 213
dragend 216
dragenter 215
DragEvent 213
dragleave 215
dragover 215
dragstart 213
drawImage 238
drop 216
dropEffect 215
duration 148
durationchange 150

E

E 80
echo 288
effectAllowed 214
elapsedTime 115
elements 164
empty 299
encodeURIComponent 310
ended 149, 150
endsWith 69
error 147, 209, 210, 319
eval 61
EvalError 98
Event 113
eventPhase 118
EventSource 318
every 74
exec 178, 179
exit 299
exp 81

F

fclose 299
File 207
file_exists 295
file_get_contents 295

FileList 207
FileReader 208, 210
files 156, 207, 219
fill 76, 227, 228
fillRect 222
fillStyle 223, 234
fillText 224
filter 74
find 74
findIndex 74
firstElementChild 126
floor 80
flush 318
focus 156, 157, 186, 187
FocusEvent 157
focusin 157
focusout 157
font 224
fopen 299
forEach 74
form 155
FormData 309
forms 124
fromCharCode 71
Function 67
fwrite 299

G

getBoundingClientRect 130
getContext 221
getData 216
getdate 288
getDate 79
getDay 79
getElementById 124
getElementsByClassName 124
getElementsByName 124
getElementsByTagName 124
getFullYear 79
getHours 79
getItem 212
getLineDash 230
getMilliseconds 79
getMinutes 79
getMonth 79
getSeconds 79
GET-параметр 293
global 181
globalAlpha 239
globalCompositeOperation 239

H

hash 191
hashchange 110, 115, 187
HashChangeEvent 115
head 123
header 299
height 130, 192, 221
host 190
hostname 191
href 190
HTML API 199, 200
HTMLAudioElement 148
HTMLBodyElement 123
HTMLCanvasElement 221
HTMLCollection 124
HTMLDocument 123
HTMLElement 125
HTMLFormElement 164
HTMLHeadElement 123
HTMLImageElement 147
HTMLInputElement 166, 167
HTMLVideoElement 148

I

ignoreCase 181
images 123
in 86
includes 69, 73
index 156
indexOf 70, 73
Infinity 22
innerHeight 185
innerHTML 137
innerText 129, 138
innerWidth 185
input 157
insertAdjacentElement 142
insertAdjacentHTML 138
insertAdjacentText 139
insertBefore 142
instanceof 86
invalid 157
isArray 72
isEqualNode 130
isFinite 61
isInteger 67
isNaN 61
isSameNode 130
isset 297

J

join 76
JSON 276, 277
json_encode 303

K

key 115
KeyboardEvent 114
keydown 109
keypress 109
keyup 109

L

lastElementChild 126
lastEventId 320
lastIndex 179
lastIndexOf 70, 73
lastModified 144
lastModifiedDate 207
left 130
length 69, 71, 82, 156, 164
lengthComputable 209
let 25, 27, 41
lineCap 229
lineJoin 229
lineTo 228
lineWidth 229
links 124
LN10 80
LN2 80
load 110, 147, 149, 186, 208, 209
loaded 209
loadeddata 150
loadedmetadata 150
loadend 209
loadstart 150, 209
localeCompare 77
localStorage 212
Location 190
log 81, 89
LOG2E 80

M

map 75
match 179, 180
Math 80
max 80
MAX_VALUE 68
measureText 224
message 98, 203, 319
min 80
MIN_VALUE 68
mousedown 109
mouseenter 109
MouseEvent 113
mouseleave 109
mousemove 109
mouseout 109
mouseover 109
mouseup 109
move_uploaded_file 298
moveTo 227
multiline 181
muted 148

N

name 97, 207
NaN 22
naturalHeight 147
naturalWidth 147
networkState 149
new 78
newURL 115
nextElementSibling 126
NodeList 124
null 83
Number 67

O

Object 82
offsetHeight 129
offsetLeft 128
offsetParent 128
offsetTop 128
offsetWidth 128
offsetX 114
offsetY 114
oldURL 115
open 272, 319
opendir 291
options 156
origin 190
outerHeight 185
outerWidth 185

P

pagehide 110, 115, 187
pageshow 110, 115, 186

PageTransitionEvent 115
pageX 113
pageXOffset 186
pageY 114
pageYOffset 186
parentElement 126
parse 277
parseFloat 61
parseInt 61
paste 157
pathinfo 294
pathname 191
patternMismatch 168
pause 150
paused 149
persisted 115
PHP 286
PI 80
play 150
playbackRate 148
playing 150
pop 72
port 191
postMessage 203
POST-параметр 296
pow 81
preventDefault 120
previousElementSibling 126
print 186
progress 150, 209, 210
ProgressEvent 209, 210
prompt 192
propertyName 115
protocol 190
prototype 247
push 72

Q

quadraticCurveTo 231
querySelector 125
querySelectorAll 125

R

random 81
RangeError 98
rangeOverflow 168
rangeUnderflow 167
ratechange 150
readAsDataURL 210
readAsText 208
readdir 291
readyState 149, 209, 273, 319
readystatechange 272
rect 232
reduce 75
reduceRight 75
ReferenceError 97
referrer 144
RegExp 171
relatedRatget 114
relatedTarget 157
reload 191
remove 136, 156
removeChild 143
removeEventListener 108, 117
removeItem 212
replace 71, 180, 191
replaceChild 143
reset 164
resize 110, 187
responseText 273
restore 240
result 208, 209, 210
return 52
reverse 77
right 130
rotate 238
round 80

S

save 240
scale 237
screen 191
Screen 191
screenLeft 185
screenTop 186
screenX 114, 186
screenY 114, 186
scripts 124
scroll 110, 187
scrollBy 186
scrollHeight 129
scrollIntoView 130
scrollLeft 129
scrollTo 186
scrollTop 129
scrollWidth 129
scrollX 186
scrollY 186
search 173, 191
seeked 150

seeking 149, 150
select 156, 157
selected 156
selectedIndex 156
selectionEnd 155
selectionStart 155
send 273, 309
sessionStorage 212
setCustomValidity 165
setData 213
setDate 79
setFullYear 79
setHours 79
setInterval 200
setItem 212
setLineDash 229
setMilliseconds 79
setMinutes 79
setMonth 79
setRequestHeader 311
setSeconds 79
setTimeout 201
shadowBlur 223
shadowColor 223
shadowOffsetX 223
shadowOffsetY 223
shift 72
shiftKey 114, 115
sin 80
sinh 81
size 207
slice 76
some 74
sort 76
source 181
splice 72
split 71, 181
sqrt 80
SQRT1_2 80
SQRT2 80
stalled 151
startsWith 69
status 273
statusText 273
stepDown 156
stepMismatch 168
stepUp 156
stopPropagation 118
Storage 212
String 69
stroke 227
strokeRect 222
strokeStyle 223, 234
strokeText 224
style 137
submit 164
substr 70
substring 70
suspend 150
SyntaxError 97

T

tagName 129
tan 81
tanh 81
target 113
terminate 204
test 173
Text 141, 156
textAlign 224
textBaseline 224
textContent 129, 138
TextMetrics 224
this 106, 107, 245
throw 98
time 287
timeupdate 150
title 144
toExponential 68
toFixed 68
toggle 136
toLocaleDateString 79
toLocaleLowerCase 69
toLocaleString 79
toLocaleTimeString 79
toLocaleUpperCase 69
toLowerCase 69
tooLong 167
top 130
toPrecision 68
toString 67
total 209
toUpperCase 69
transitionend 109, 115
TransitionEvent 115
translate 237
trim 69
trunc 80
type 113, 207
TypeError 97
typeMismatch 167
typeof 29

U

undefined 29
unlink 300
unshift 72
URIError 98
URL 144

V

valid 167
validationMessage 166
validity 167
ValidityState 167
value 155
valueMissing 167
var 27, 41
videoHeight 148
videoWidth 148
volume 148
volumechange 150

W

waiting 150
wheel 109, 114
WheelEvent 114
which 114, 115
width 130, 192, 221, 224
willValidate 167
Window 123, 185
Worker 202
write 139
writeln 139

X

x 130
XAMPP 329
◊ панель управления 334
XML 271
XMLHttpRequest 272

Y

y 130

А

Атрибут тега
◊ draggable 213
◊ height 221
◊ id 104
◊ pattern 182
◊ script 35
◊ src 60
◊ width 221

Б

Библиотека 60
Блок 40
Бэкенд 302

В

Валидация 165
Вариант 175
Веб-приложение
◊ клиентское 160
◊ серверное 160, 285
Веб-сайт: сверхдинамический 274
Веб-служба 302
Веб-страница
◊ по умолчанию 292
◊ серверная 289
Веб-сценарий 35
◊ внешний 60
◊ внутренний 35
Возврат результата 20
Возобновление 95
Временна́я отметка 78
◊ в стиле UNIX 287
Выражение 35
◊ блочное 40
◊ выбора 41
◊ условное 38
◊ условное в сокращенной форме 39
◊ условное множественное 39
Выход из функции 94

Г

Геттер 251
Градиент 233
◊ линейный 233
◊ радиальный 235
Графическая закраска 236
Графический контекст 221
Группа 177

Д

Декремент 28

Ж

Журнал 336

З

Заголовок 299
Замыкание 266
Застревание в кеше 338

И

Имя
◊ класса 66
◊ переменной 24, 287
◊ события 105
◊ функции 51
Индекс 61
Инициализация 261
Инкремент 28
Инспектор DOM 95
Исключение 96
◊ обработка 97
◊ обработчик 97

К

Квантификатор 175
Кеш 338
Класс 66
◊ базовый 83
◊ производный 83
Ключ 288
Ключевая точка 233
Коллекция 82
Комментарий 48
Компонент 261
◊ невидимый 280
Конкатенация 23
Консоль 17, 89
Константа 27
Конструктор 245
Контекст исполнения 127
Кривая Безье 231
◊ квадратическая 231
◊ кубическая 231

Л

Лайтбокс 258
Литерал
◊ регулярного выражения 171
◊ строковый 22, 287
Лог 336
Логическое
◊ И 24
◊ ИЛИ 24, 56
◊ НЕ 24
◊ отрицание 24
◊ сложение 24, 56
◊ умножение 24

М

Маршрутизация 309
Маска 240
Массив 61
◊ ассоциативный 288
◊ вложенный 63
◊ внешний 63
◊ размер 61
◊ элемент 61
Межпоточное сообщение 203
Метасимвол 173
Метод 66
◊ переопределение 255
◊ статический 67, 257
Модификатор 171

Н

Наследование 83, 253
Нулевая ссылка 83

О

Обработчик события 105, 107, 139
◊ по умолчанию 119
◊ привязка 106, 108
◊ удаление 108

Обратная ссылка 177
Объект 65
◊ служебный 82
◊ текущий 66
Объектная нотация 82
Окружность
◊ конечная 235
◊ начальная 235
Операнд 20
Оператор 20
◊ арифметический 21, 28
◊ бинарный 21
◊ возврата результата 52
◊ вывода 288
◊ вызова функции 53
◊ генерирования исключения 98
◊ доступа к элементу массива 62, 66, 70, 82
◊ доступа к элементу объекта 65
◊ комбинированного присваивания 26
◊ конкатенации строк 23, 289
◊ логический 24
◊ объявления переменных 25, 27
◊ получения типа 29
◊ прерывания 42, 47
▫ прохода 47
◊ принадлежности классу 86
◊ присваивания 25
◊ проверки существования свойства 86
◊ создания объекта 78
◊ сравнения 23
◊ тернарный 28
◊ удаления свойства 85
◊ унарный 21, 28
◊ условный 28
Отладчик 90

П

Параметр функции 51
◊ необязательный 55
Переменная 24
◊ глобальная 54
◊ локальная 54
◊ обращение 26
◊ объявление 25, 27, 41, 287
Перетаскивание 213
◊ источник 213
◊ приемник 213
Перо 227
Подквантификатор 177
Подкласс 83
Поднабор 174
Поиск
◊ без учета регистра 172
◊ глобальный 179
◊ жадный режим 176
◊ многострочный 180
◊ обычный 178
◊ щедрый режим 177
Поток 202
◊ основной 202
◊ фоновый 202
Преобразование 237
◊ типов 30
Приоритет оператора 21
Присваивание 25
Прототип 246
Прохождение 116

Р

Регулярное выражение 171
Рекурсия 57
◊ бесконечная 58

С

Свойство 65, 252
◊ динамическое 251, 252
◊ добавленное 85
◊ статическое 68, 256
Серверное сообщение 317
Сеттер 251
Слайдер 103
Событие 105
◊ всплывающее 116
◊ источник 113
◊ невсплывающее 117
Состояние холста 240
Стек вызовов 58
Строгий режим 48
Строка 20, 22
Суперкласс 83

Т

Таймер
◊ однократный 201
◊ периодический 199
Тег
◊ ?php 287
◊ canvas 221
◊ script 35

Тип данных 20
◊ значащий 84
◊ логический 20
◊ неопределенный 29
◊ объектный 63, 83
◊ ссылочный 84
◊ строковый 20
◊ функциональный 60
◊ числовой 20
Точка останова 90, 92, 95
Трассировка 90

У

Унарный минус 28
Условие 23

Ф

Фаза 116
◊ всплытия 116
◊ источника 115, 116
◊ погружения 116
Файл веб-сценария 60
◊ привязка 60
Фреймворк 325
Фронтенд 301
Функция 51
◊ анонимная 58
◊ вложенная 54
◊ встроенная 61
◊ вызов 52, 53
◊ именованная 58
◊ объявление 51
◊ стрелка 59, 107

Х

Хеш 288
Холст 221
Хранилище 212
◊ постоянное 212
◊ сессионное 212

Ц

Цикл 43
◊ итерация 43
◊ по свойствам 87
◊ проход 43
◊ с постусловием 47
◊ с предусловием 46
◊ со счетчиком 43
◊ счетчик 43, 46
◊ условный 43

Ч

Число 20
◊ вещественное 20
◊ с плавающей точкой 20

Ш

Шаг
◊ с заходом 93
◊ с обходом 94

Э

Элемент веб-страницы: базовый 261

Я

Якорь 104

JavaScript

20 уроков для начинающих

Дронов Владимир Александрович, профессиональный программист, писатель и журналист, работает с компьютерами с 1987 года. Автор более 30 популярных компьютерных книг, в том числе «HTML и CSS: 25 уроков для начинающих», «Django 2.1. Практика создания веб-сайтов на Python», «HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера», «Python 3. Самое необходимое», «Python 3 и PyQt 5. Разработка приложений», «Laravel. Быстрая разработка современных динамических Web-сайтов на PHP, MySQL, HTML и CSS», книг по продуктам Adobe Flash и Adobe Dreamweaver различных версий. Его статьи публикуются в журналах «Мир ПК» и «ИнтерФейс» (Израиль) и интернет-порталах «IZ City» и «TheVista.ru».

Простым языком, кратко, наглядно, в картинках рассказано о программировании веб-сценариев на языке JavaScript. В книге 20 иллюстрированных уроков, 40 практических упражнений и более 70 заданий для самостоятельной работы.

JavaScript
в картинках и упражнениях

Вы узнаете, как

- обрабатывать события;
- добавлять элементы на веб-страницу;
- создать свой видеопроигрыватель;
- проверить правильность введения данных в веб-форму;
- выполнять длительные вычисления в фоновом потоке;
- нарисовать график на веб-странице;
- объявить свой класс;
- написать компонент;
- загрузить данные с сервера;
- программировать фронтенды и бэкенды;
- отлаживать веб-сценарии.

Все необходимые для работы файлы и результаты выполнения упражнений можно скачать по ссылке ftp://ftp.bhv.ru/9785977565899.zip, а также со страницы книги на сайте www.bhv.ru.

bhv®

191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru

ISBN 978-5-9775-6589-9