

Никольский А.П., Дубовик Е.В.

Справочник

JavaScript

КРАТКО # БЫСТРО # ПОД РУКОЙ



Синтаксис JavaScript * Объекты и события JS
Объектная модель браузера * Библиотеки jQuery и jQuery UI
Примеры использования

Никольский А.П., Дубовик Е.В.

Справочник JavaScript

КРАТКО # БЫСТРО # ПОД РУКОЙ



**"Наука и Техника"
г. Санкт-Петербург**

УДК 004.42 ББК 32.973

ISBN 978-5-94387-333-1

Никольский А.П., Дубовик Е.В.

Справочник JavaScript. Кратко, быстро, под рукой — СПб.: Наука и Техника, 2021. — 304 с., ил.

Серия «Справочник»

Данный справочник содержит всю ключевую информацию о **JavaScript** в удобной и наглядной форме. Структура справочника позволяет быстро и удобно находить нужную информацию, получать примеры использования тех или иных элементов и конструкций **JavaScript**.

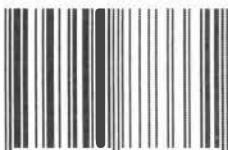
Отдельное внимание уделено отраслевым рекомендациям по хорошему стилю программирования на **JavaScript**, объектной модели современных браузеров, отладке программ и обработке ошибок.

Справочник будет полезен всем, кто использует или изучает **JavaScript**: от начинающих до профессионалов.

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Издательство не несет ответственности за возможный ущерб, причиненный в ходе использования материалов данной книги, а также за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-5-94387-333-1



9 78-5-94387- 333-1

Контактные телефоны издательства:

(812) 412 70 26

Официальный сайт: www.nit.com.ru

© Никольский А.П.

© Наука и Техника (оригинал-макет)

Содержание

ВВЕДЕНИЕ.....	11
ГЛАВА 1. БАЗОВЫЕ ПОЛОЖЕНИЯ JAVASCRIPT	13
1.1. НЕ ЗАБЫВАЕМ О СКРИПТИНГОВОЙ ПРИРОДЕ	14
1.2. ОБЪЕКТНАЯ МОДЕЛЬ ДОКУМЕНТА	17
1.3. ПЕРВАЯ ПРОГРАММА	19
1.4. КОММЕНТАРИИ В JAVASCRIPT	22
1.5. ДИАЛОГОВЫЕ ОКНА.....	23
1.5.1. Метод alert() – простое модальное окно с сообщением и кнопкой "OK".....	24
1.5.2. Метод confirm() - окно с кнопками OK и Cancel.....	25
1.5.3. Метод prompt() - диалоговое окно для ввода данных	26
1.6. СПЕЦИАЛЬНЫЕ СИМВОЛЫ.....	28
1.7. КЛЮЧЕВЫЕ СЛОВА.....	29
ГЛАВА 2. СИНТАКСИС JAVASCRIPT	31
2.1. ПЕРЕМЕННЫЕ	32
2.1.1. Особенности объявления переменных.....	32
2.1.2. Типы данных и преобразование типов.....	34
2.1.3. Области видимости переменной	38
2.2. ОПЕРАТОРЫ И ВЫРАЖЕНИЯ В JAVASCRIPT	40
2.2.1. Различные типы выражений	40
2.2.2. Присваивание переменным значений	41
2.2.4. Булевые операторы	42
2.2.5. Операторы сравнения	43
2.2.6. Двоичные операторы	44

2.2.7. Конкатенация	44
2.2.8. Приоритет выполнения операторов	44
2.3. ОСНОВНЫЕ КОНСТРУКЦИИ ЯЗЫКА	46
2.3.1. Условный оператор if	46
2.3.2. Оператор выбора switch.....	49
2.3.3. Циклы	51
Цикл со счетчиком	52
Цикл while	53
Цикл do..while.....	54
Управление выполнением цикла.	
Операторы break и continue.....	54
Вложенность циклов	55
ГЛАВА 3. МАССИВЫ В JAVASCRIPT.....	57
3.1. ВВЕДЕНИЕ В МАССИВЫ	58
3.2. ИНИЦИАЛИЗАЦИЯ МАССИВА.....	59
3.3. ИЗМЕНЕНИЕ И ДОБАВЛЕНИЕ ЭЛЕМЕНТОВ МАССИВА.....	60
3.4. МНОГОМЕРНЫЕ МАССИВЫ	60
3.5. ПРИМЕР ОБРАБОТКИ МАССИВА	61
ГЛАВА 4. ФУНКЦИИ В JAVASCRIPT	65
4.1. ОСНОВНЫЕ ПОНЯТИЯ.....	66
4.1.1. Способы объявления функций	66
4.1.2. Практические примеры.....	68
4.2. РАЗМЕЩЕНИЕ ФУНКЦИЙ ВНУТРИ СЦЕНАРИЯ	70
4.3. РЕКУРСИЯ	72
4.4. ОБЛАСТЬ ВИДИМОСТИ ПРИ ИСПОЛЬЗОВАНИИ ФУНКЦИЙ	73

ГЛАВА 5. ОТЛАДКА ПРОГРАММЫ И ОБРАБОТКА ОШИБОК	77
5.1. КАК УВИДЕТЬ СООБЩЕНИЕ ОБ ОШИБКЕ	78
5.2. СИНТАКСИЧЕСКИЕ ОШИБКИ.....	81
5.3. ЛОГИЧЕСКИЕ ОШИБКИ	82
5.4. RUN-TIME ERRORS ИЛИ ОШИБКИ ВРЕМЕНИ ВЫПОЛНЕНИЯ....	83
5.5. КОНСТРУКЦИЯ TRY	84
5.6. МЕТОД CONSOLE.LOG()	85
ГЛАВА 6. ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ	87
6.1. ОСНОВНЫЕ КОНЦЕПЦИИ	88
6.1.1. Введение в ООП	88
6.1.2. Объявление класса.....	92
6.1.3. Выражение класса	93
6.2. ТЕЛО КЛАССА	94
6.2.1. Конструктор.....	94
6.2.2. Статические методы и свойства. Ключевое слово static	94
6.2.3. Использование this в прототипных и статических методах	95
6.2.4. Свойства экземпляра	96
6.2.5. Публичные и приватные поля	96
6.2.6. Наследование. Оператор extends	97
6.3. СОЗДАНИЕ ПОЛЬЗОВАТЕЛЬСКИХ ОБЪЕКТОВ	98
6.4. ПРОТОТИПЫ	101
6.5. ПРОСТРАНСТВА ИМЕН	102

ГЛАВА 7. ВСТРОЕННЫЕ ОБЪЕКТЫ JAVASCRIPT	105
 7.1. ОБЪЕКТ GLOBAL	106
 7.2. ОБЪЕКТ NUMBER	108
 7.3. ОБЪЕКТ STRING.....	109
 7.4. ОБЪЕКТ ARRAY	112
7.4.1. Свойства и методы объекта	112
7.4.2. Сортировка массива.....	115
7.4.3. Многомерные массивы.....	116
7.4.4. Ассоциативные массивы.....	116
 7.5. ОБЪЕКТ MATH	117
 7.6. ОБЪЕКТЫ FUNCTION И ARGUMENTS	119
 7.7. ОБЪЕКТ DATE	120
 7.8. ОБЪЕКТ REGEXP	123
ГЛАВА 8. СОБЫТИЯ В JAVASCRIPT	127
 8.1. ЧТО ТАКОЕ СОБЫТИЕ?	128
 8.2. СОБЫТИЯ МЫШИ	129
 8.3. СОБЫТИЯ КЛАВИАТУРЫ	130
 8.4. СОБЫТИЯ ДОКУМЕНТА	130
 8.5. СОБЫТИЯ ФОРМЫ	131
 8.6. ПОСЛЕДОВАТЕЛЬНОСТЬ СОБЫТИЙ	131
 8.7. ВСПЛЫВАНИЕ СОБЫТИЙ	133
 8.8. ДЕЙСТВИЕ ПО УМОЛЧАНИЮ	135
 8.9. ОБРАБОТЧИКИ СОБЫТИЙ	135
 8.10. ОБЪЕКТ EVENT	137

ГЛАВА 9. ОБЪЕКТНАЯ МОДЕЛЬ БРАУЗЕРА	141
9.1. СТРУКТУРА ОБЪЕКТНОЙ МОДЕЛИ	142
9.2. ОСНОВНЫЕ ОБЪЕКТЫ ОБЪЕКТНОЙ МОДЕЛИ IE/CHROME	143
9.3. ОБЪЕКТ WINDOW	144
9.3.1. Метод open(): создаем новые окна	148
9.3.2. Отображение модального диалога.....	150
9.3.3. Метод setTimeout()	153
9.4. ОБЪЕКТ NAVIGATOR: ПОЛУЧЕНИЕ ИНФОРМАЦИИ О БРАУЗЕРЕ И СИСТЕМЕ.....	156
9.5. ОБЪЕКТ SCREEN: ИНФОРМАЦИЯ О МОНИТОРЕ ПОЛЬЗОВАТЕЛЯ	158
9.6. ОБЪЕКТ LOCATION: СТРОКА АДРЕСА ПОЛЬЗОВАТЕЛЬСКОГО АГЕНТА	159
9.7. ОБЪЕКТ HISTORY: СПИСОК ИСТОРИИ	160
9.8. ОБЪЕКТ DOCUMENT: ДОСТУП К ЭЛЕМЕНТАМ ДОКУМЕНТА ...	161
9.9. ОБЪЕКТ STYLE: ДОСТУП К ТАБЛИЦЕ СТИЛЕЙ	167
9.10. ОБЪЕКТ SELECTION: ВЫДЕЛЕНИЕ ТЕКСТА	168
9.11. НЕМНОГО ПРАКТИКИ. РАБОТА С COOKIES	170
9.11.1. Добавление сайта в Избранное.....	170
9.11.2. Установка сайта в качестве домашней страницы.....	171
9.11.3. Работа с Cookies	171
ГЛАВА 10. РАБОТА С ФОРМАМИ.....	175
10.1. КОЛЛЕКЦИЯ FORMS	176
10.2. СВОЙСТВА, МЕТОДЫ И СОБЫТИЯ ОБЪЕКТА ФОРМЫ.....	177

10.3. ПОЛУЧЕНИЕ ДАННЫХ ИЗ ПОЛЯ ВВОДА. ПРОВЕРКА ПРАВИЛЬНОСТИ ВВОДА	178
10.4. РАБОТА С TEXTAREA	179
10.5. РАБОТА С ФЛАЖКАМИ	181
10.6. РАБОТА С КНОПКАМИ	183
10.7. ПРОВЕРКА ПРАВИЛЬНОСТИ E-MAIL.....	184
ГЛАВА 11. БИБЛИОТЕКА JQUERY	191
11.1. ПОДКЛЮЧЕНИЕ БИБЛИОТЕКИ JQUERY	192
11.2. ВЫБОР ЭЛЕМЕНТА HTML. СЕЛЕКТОРЫ	194
11.2.1. Выбор всех элементов	194
11.2.2. Выбор элемента по его id.....	195
11.2.3. Поиск всех элементов определенного типа.....	196
11.2.4. Поиск элементов по имени класса	197
11.2.5. Селекторы + и ~	198
11.3. ФИЛЬТРЫ.....	199
11.3.1. Фильтр :first	200
11.3.2. Выбор последнего ряда таблицы	201
11.3.3. Выбор четных и нечетных элементов	202
11.3.4. Другие фильтры	203
11.4. РАБОТА С АТРИБУТАМИ ЭЛЕМЕНТОВ.....	206
11.4.1. Методы attr() и removeAttr(): получение значения, установка и удаление атрибутов	206
11.4.2. Методы addClass(), removeClass(): работа со стилями ..	208
11.4.3. Методы html() и text(): работа с содержимым элементов	212
11.4.4. Метод val(): работа с атрибутом value	213

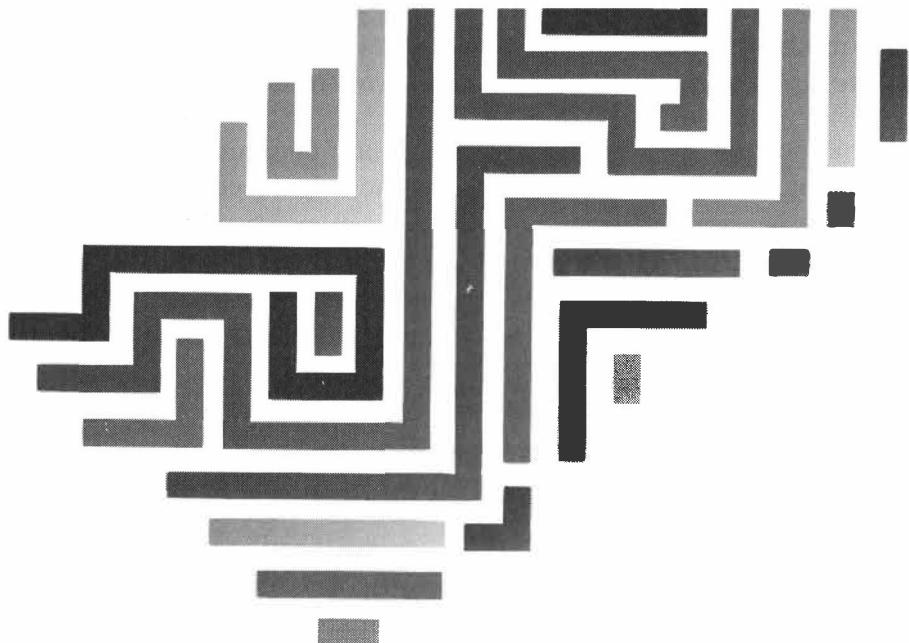
11.5. ВИЗУАЛЬНЫЕ ЭФФЕКТЫ.....	216
11.5.1. Методы hide(), show(), toggle(): управление видимостью	216
11.5.2. Методы slideUp(), slideDown() и slideToggle(): скольжение	217
11.5.3. Методы fadeOut(), fadeIn() и fadeToggle(): эффект затухания.....	219
11.5.4. Метод fadeTo(): плавное изменение прозрачности	220
11.5.5. Метод animate(): простейшая анимация.....	222
11.6. РАБОТА С CSS	224
11.6.1. Метод css(): получение и установка значения CSS-свойства	224
11.6.2. Другие методы.....	227
11.7. РАБОТА С СОДЕРЖИМЫМ ЭЛЕМЕНТА.....	230
11.7.1. Операции над содержимым элементов.....	230
11.7.2. Методы appendTo() и prependTo()	231
11.7.3. Методы after() и before().....	232
11.7.4. Методы wrapInnner() и wrapAll().....	233
11.7.5. Поиск и замена элементов.....	236
11.7.6. Другие полезные методы.....	237
11.8. ОБРАБОТКА СОБЫТИЙ.....	238
11.9. AJAX И JQUERY	240
11.9.1. Загрузка страницы. Метод load()	241
11.9.2. Получение данных от сервера с использованием запроса POST	244
11.9.3. Получение данных от сервера с использованием запроса GET.....	246
11.9.4. Загрузка и выполнение JS-файлов.....	247
11.9.5. Получение JSON-данных.....	249

ГЛАВА 12. БИБЛИОТЕКА JQUERY UI	253
12.1. ЗНАКОМСТВО С БИБЛИОТЕКОЙ	254
12.1.1. Что такое jQuery UI	254
12.1.2. Что представляют собой плагины jQuery UI	255
12.1.3. Преимущества jQuery UI.....	257
12.1.4. Использование jQuery UI	258
12.2. ВИДЖЕТЫ JQUERY UI.....	259
12.2.1. Выбор даты.....	259
12.2.2. Диалоговое окно.....	263
12.2.3. Раскрывающиеся секции	264
12.2.4. Индикатор процесса	267
12.2.5. Вкладки	270
12.2.6. Автозавершение	273
12.2.7. Кнопка	275
12.2.8. Меню	277
12.2.9. Эффектная подсказка ToolTip	282
12.2.10. Ползунок для ввода целых значений	283
12.3. АНИМАЦИОННЫЕ ЭФФЕКТЫ JQUERY UI.....	287
12.3.1. Как применить эффект.....	288
12.3.2. Эффекты и их параметры.....	289
12.3.3. Демонстрация всех эффектов сразу	293
12.4. МАНИПУЛЯЦИЯ ЦВЕТОМ	303

Благодарности

Данная книга написана сотрудниками компании ООО «Цифровые Бизнес-Платформы». Авторы выражают благодарность всем своим коллегам, при содействии и с участием которых был создан данный справочник.

ВВЕДЕНИЕ



JavaScript – это язык программирования, позволяющий сделать ваши страницы "живыми", то есть динамическими. Кроме всевозможных эффектов, JavaScript позволяет автоматизировать многие вещи. Если, скажем, лет 15 назад можно было встретить сайт, разработанный на чистом HTML (пусть и с использованием таблиц стилей), то сейчас очень сложно себе представить страничку, не использующую JavaScript.

Интересно, что первые версии этого языка назывались LiveScript, подчеркивая основное назначение языка – "оживлять" элементы страницы, саму страницу. Однако в то время был очень популярен язык Java, а поскольку синтаксис JavaScript очень напоминает синтаксис Java, было решено назвать его именно JavaScript – чтобы привлечь больше внимания.

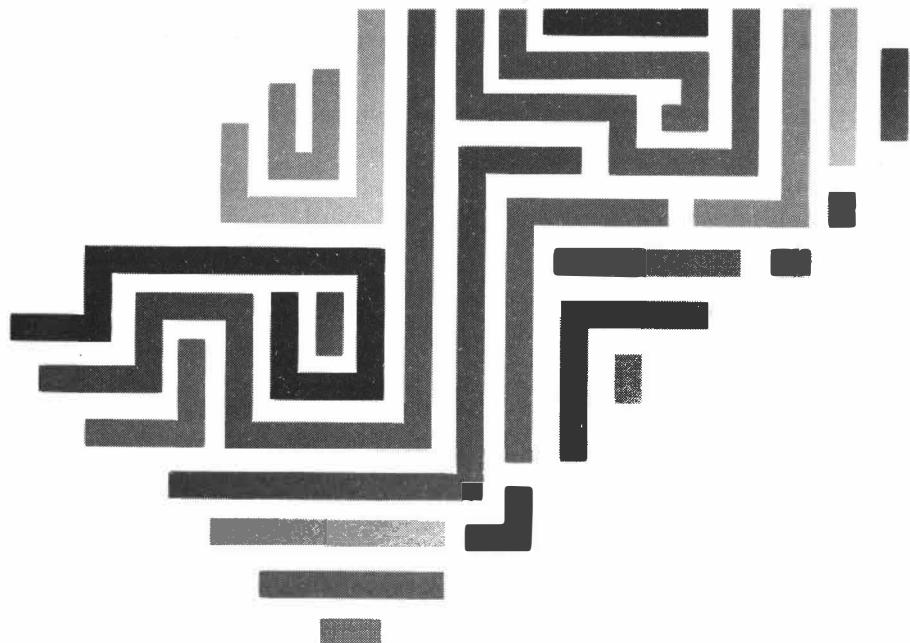
Код JavaScript находится в составе HTML-документа (или выносится в отдельный файл, если код слишком большой). Сам же JavaScript также может генерировать HTML-код в случае необходимости.

Нужно понимать, что JavaScript-сценарии в большинстве случаев выполняются на стороне клиента – прямо в браузере. В этом есть, как преимущества, так и недостатки. К преимуществам можно смело отнести снижения нагрузки на сервер, поскольку существенная доля обработки происходит на стороне клиента и именно ресурсы клиента тратятся на все это. Недостаток – не всегда один и тот же скрипт одинаково хорошо работает во всех браузерах. В последнее время с кроссбраузерной совместимостью возникает все меньше и меньше проблем, но о ней нужно помнить при разработке собственных скриптов.

Данная книга сочетает в себе, как теорию, так и практику. Будет описан синтаксис JavaScript, а также будет приведен ряд уже готовых рецептов.

ГЛАВА 1.

Базовые положения JavaScript



1.1. Не забываем о скриптинговой природе

Хотя в названии изучаемого языка и есть слово "Java", нужно понимать, что природа этих двух языков программирования совершенно иная. По сути, JS только отдаленно напоминает Java благодаря схожести синтаксиса. Но сравнивать эти два языка так же некорректно, как сравнивать самолет с автобусом. И то, и другое является транспортным средством (является языком программирования), и то и другое позволяет перевозить пассажиров (может использоваться в веб-страницах), но на этом схожесть заканчивается.

Даже если сравнивать синтаксис языков, то JS похож лишь "если один глаз закрыть, а другой прищурить". Если Java является полноценным объектно-ориентированным языком программирования, то в JS нет полноценной поддержки ООП, а классы – это всего лишь "синтаксический сахар" (надстройка) над объектами.

Природа Java и JS – разная. JS – это небольшие программы, встроенные в веб-страницу (как в коде самой страницы, так и в отдельном файле). При желании, разработчик может просмотреть код любой программы, написанной на JS. А вот с Java ситуация иная. Java-программы (их еще называют апплетами) хранятся на сервере отдельно, в виде бинарных файлов.

- Java является компилируемым языком программирования, то есть написанная программа компилируется в бинарный файл,

который потом можно встроить в веб-страницу (сейчас мы говорим только о веб-страницах, но не забываем, что Java – это кроссплатформенный язык и на нем можно написать программы для любой операционной системы).

- JS, как и любой скрипtingовый язык, является интерпретируемым, то есть он не компилируется в какой-то бинарный файл, который будет автономно "лежать" на сервере. JS-код постоянно находится в текстовом виде и интерпретируется браузером пользователя. Любую JS-программу можно просмотреть без применения специального ПО вроде дизассемблера.

Примечание. Из соображений оптимизации передаваемых данных часто JavaScript-код подлежит минимизации – процессу, когда из него удаляются все незначимые пробельные символы (удаление которых не повредит функционалу) и комментарии. В этом случае код становится нечитабельным, но благодаря функции **Pretty Print** (находится на вкладке **Source** и обозначена {}) средств разработчика – это не проблема (рис. 1). Эта функция вам очень пригодится при исследовании чужого кода, находящегося на сайте, доступа

The screenshot shows two side-by-side code snippets. On the left, the original complex, unformatted code is shown. It includes numerous nested if statements, loops, and various JavaScript constructs like closures and object literals. On the right, the same code is presented after being run through the Pretty Print function. The result is significantly more readable, with each line of code having its own space and indentation. The code is organized into several functions, each starting with a colon (:) and containing multiple statements. The overall structure is much clearer than the original, though the logic remains identical.

Рис. 1.1. Использование функции **Pretty Print** – слева – до, справа – после

к исходникам которого у вас нет. Попросту говоря – вам понравилась какая-то функция на чужом сайте, и вы хотите реализовать аналогичную на своем. Тогда вам достаточно посмотреть код JavaScript, адаптировать его под свой сайт и вы получите то, что хотите.

Изначально JS был разработан для увеличения функциональности HTML-страниц. Другими словами, на нем можно реализовать те "фишки", которые нельзя реализовать с помощью стандартных тегов HTML. Современные стандарты HTML, конечно, предлагают более совершенный функционал, но если HTML просто развивается, то JS развивается семимильными шагами. Приведем простой пример. Относительно недавно в HTML появилась возможность проверки валидности электронного адреса. Ранее эта задача решалась исключительно на JS. Но если HTML научился только проверять правильность e-mail, то JS скоро научится запускать космические ракеты.

Программа, написанная на JS, имеет доступ к свойствам документа (веб-страницы) и самого браузера. На JS вы можете просмотреть, какой URL введен в адресную строку браузера, изменить заголовок окна, узнать, произошло ли какое-то событие (например, пользователь нажал на кнопку или открыл выпадающий список) и т.д. Описывать все возможности JS прямо сейчас нет смысла – вы познакомитесь с ними далее в этой книге.

В отличие от Java, разработчику не нужен какой-либо специальный софт для разработки. Все, что нужно – обычный текстовый редактор. Если не брать уж совсем простой Блокнот, мы можем порекомендовать следующие редакторы:

- **NotePad2** – очень простой редактор, но поддерживающий подсветку синтаксиса, преобразование кодировок (очень важная функция) и нумерацию строк. Для простых JS-скриптов его возможностей будет вполне достаточно.

- **Atom** – кроссплатформенный редактор с огромным функционалом, расширяемым дополнительными плагинами. Иногда кажется, что его возможности безграничны и скоро Atom начнет писать код за программиста. Впрочем, автодополнение кода у него уже есть! К недостаткам можно отнести нерасторопность редактора – он медленный по сравнению с тем же MS Visual Code.
- **MS Visual Code** – отличный редактор кода от Microsoft и при этом совершенно бесплатный. Иногда кажется, что это что-то из области научной фантастики – Microsoft и бесплатный продукт. Тем не менее, это так. Учитывая, что этот редактор более шустрый, чем Atom – это наш выбор.



Рис. 1.2. Редактор Microsoft Visual Studio Code

1.2. Объектная модель документа

Каждый контейнер языка HTML можно рассматривать как совокупность свойств, методов и событий. Со всеми этими свойствами,

методами и событиями может работать JS. Например, вы можете разработать кнопку и установить для нее обработчик, который будет запущен при нажатии на эту кнопку. В свою очередь этот обработчик может изменить свойства другого объекта документа, например, изменить цвет блока фона `div`.

В этой главе мы не будем вникать в ее подробности, а желающие "бежать впереди паровоза" могут узнать больше по следующей ссылке:

https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Details_of_the_Object_Model

Пока вам нужно знать, что есть старший класс `Window`, позволяющий обращаться к методам и свойствам HTML-страницы и браузера. Например, метод `close()` позволяет закрыть окно браузера, а свойство `location` - обратиться к адресной строке браузера.

Новичкам, не знакомым с концепцией ООП, термины "метод" и "объект" мало о чем говорят. Не расстраивайтесь, пока принимайте все как есть, а дальше вы поймете, что к чему. В JavaScript все основано на объектах и прототипах (поскольку это объектно-ориентированный язык, иногда можно встретить формулировку прототипно-ориентированный язык) и без них вы не сможете написать свои программы. Именно поэтому мы начали разговор об объектной модели JavaScript в этой вводной главе.

Объект можно воспринимать как совокупность данных и методов (функций) для их обработки. В JavaScript с некоторыми объектами также связываются определенные события.

Попробуем "объектизировать" человека. Создадим объект с именем `Human`. У такого объекта может быть масса характеристик - имя, пол, дата рождения и т.д. Все это называется свойствами объекта. Обратиться к свойству можно так:

`объект.свойство`

Например:

```
Human.Sex = 'M';
Human.Name = 'Max';
Human.YearOfBirth = '1988';
```

Для обработки данных объекта используется функция, называемая методом. Например, мы можем написать функцию, которая возвращает пол человека – GetSex(). Вызвать метод можно так:

```
объект.метод(параметры);
```

Например:

```
Human.GetSex();
```

Сопоставить метод с функцией можно так:

```
объект.метод = имя_функции;
```

С объектом можно связать какое-нибудь событие. Например, когда человек умирает, вызывается событие OnDeath. Что будет делать обработчик этого события (функция, сопоставленная событию и вызывающаяся для обработки события)? Зависит от задач, как правило, в подобных случаях она может освобождать занятые ресурсы, удалять запись из БД о человеке и т.д.

1.3. Первая программа

Чтобы ваша JavaScript-программа (или сценарий) запустилась, ее нужно внедрить в HTML-код документа (или "связать" программу с документом). Обратите внимание - я использовал два разных термина - внедрение и связка. Я сделал это умышленно, потому что есть разные способы взаимного существования JavaScript и HTML.

Самый простой из них - использование тега <SCRIPT>, который обычно расположен до тега </BODY> (скрипты, в отличие от CSS, принято размещать в конце страницы, до закрывающегося тега </BODY>), например:

```
<SCRIPT>
document.write("<h1>Привет, читатель!</h1>");
</SCRIPT>
```

Данный сценарий будет выполнен при открытии страницы и выведет строку:

```
<h1>Привет, читатель!</h1>
```

Конечно, далеко не всегда нужно, чтобы скрипт запускался при открытии страницы. Часто нужно, чтобы код запускался при определенном событии, например, когда изменился размер окна или была нажата кнопка. Рассмотрим пример, отображающий модальное диалоговое окно при нажатии кнопки:

```
<INPUT TYPE="button" VALUE="Показать" onClick="window.
alert('Привет!')">
```

Поскольку программа небольшая, мы не стали описывать ее в теге <script>, а описали прямо в атрибуте onClick для нашей кнопки.

Не всегда обработчик события компактный. Иногда он просто не помещается в одну-две строчки. Конечно, максимальная длина значения атрибута HTML-тега составляет 1024 символа, что вполне хватило бы для несложных обработчиков. Однако такие длинные обработчики значительно ухудшают читабельность HTML-кода, поэтому такие обработчики мы можем оформить в виде функции, например:

```
<form>
    <input type="button" value="Показать"
onClick="openAlert()">
</form>
```

Функция openAlert() должна быть определена в теге <SCRIPT>. Полный код (HTML и JavaScript) нашего первого "проекта" (да, первая программа в этой книге не будет выводить фразу "Привет, мир!") приведен в лист. 1.1.

Листинг 1.1. Наш первый JavaScript-проект

```
<html>
  <head>
    <title>Первая программа на JavaScript</title>
  </head>
  <body>
    <form>
      <input type="button" value="Показать"
onClick="openAlert()">
    </form>
    <script language="JavaScript">
      function openAlert() {
        window.alert('Привет! ');
      }
    </script>
  </body>
</html>
```

Данная страница отобразит кнопку **Показать**, при нажатии которой будет отображено модальное окно с надписью "Привет".

Здесь мы указали атрибут **language** и явно указали, что наша программа написана на JS (хотя ее еще можно написать на Visual Basic Script). Современные браузеры не требуют указания этого атрибута – считается, что если он не указан, то код написан на JavaScript, поэтому далее мы не будем его указывать, как и атрибут **type="text/javascript"** – для современных браузеров он не есть обязательным (настолько JS стал популярным, что вытеснил все остальные языки).

Примечание. Для атрибута **language** доступны следующие значения: "JavaScript", "JScript", "VBS". С первым и последним все понятно, а вот второй – это язык JScript, который явля-

ется разновидностью JS, но от компании Microsoft. Данный язык не получил должного распространения.

С помощью атрибута **src** можно указать файл, в котором находится сценарий. Код, занимающий большой объем, например, сотни строк, принято выносить в отдельный файл с расширением .js, чтобы не захромождать код веб-страницы. Подключить внешний .js-файл можно посредством атрибута **src**:

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

Следующий атрибут тега <SCRIPT> - **defer**. Данный атрибут позволяет отложить выполнение сценария до полной загрузки страницы (обычно выполнение скрипта начинается сразу после загрузки страницы):

```
<script src="main.js" defer>
// некоторый код, выполнение которого можно отложить
</script>
```

Пятый атрибут, который вам придется использовать для оптимизации – **async**. Он означает, что код скрипта должен выполняться асинхронно, что позволяет уменьшить время загрузки страницы. Инструмент PageSpeed Insights от Google настоятельно рекомендует загружать скрипты с атрибутом **async**, но вы должны понимать, что не каждый скрипт может корректно работать с этим атрибутом и его использование в некоторых случаях означает, что вы можете потерять необходимый функционал.

1.4. Комментарии в JavaScript

В этой главе вы уже успели познакомиться с комментариями, во всяком случае, неявно. Теперь настало время познакомиться с ними явно. Комментарии в JS могут быть одностroчными и многостроч-

ными. Однострочный комментарий начинается с двух знаков //, с этим типом комментариев вы уже знакомы:

```
// комментарий  
i++; // увеличиваем i
```

Комментарием является все, что находится после // и до конца строки.

Многострочный комментарий начинается символами /* и заканчивается символами */, например:

```
/* это пример  
многострочного  
комментария */
```

Какие комментарии использовать - зависит от вас. Однострочные комментарии удобно использовать для комментирования отдельных строчек кода. Многострочные комментарии подойдут для объяснения того, что делает целый блок, например, описать, что делает функция и какие параметры ей нужно передать.

1.5. Диалоговые окна

Для взаимодействия с пользователем, то есть для ввода данных и вывода результатов работы программы, как правило, используются HTML-формы и возможность вывода HTML-кода прямо в документ (метод document.write). Этот способ удобен тем, что вы, используя HTML и CSS, можете оформить форму ввода данных так, как вам благорассудится. То же самое можно сказать и о выводе данных. Из JS-сценария вы можете выводить любой HTML-код, позволяющий как угодно оформить вывод.

Но в некоторых ситуациях этих возможностей оказывается очень много. Иногда нужно просто вывести диалоговое сообщение, например, сообщить пользователю, о том, что введенный им пароль слишком простой или слишком короткий. В этом разделе мы разберемся, как выводить диалоговые окна, позволяющие выводить короткие сообщения (например, сообщения об ошибках ввода) и обеспечивающий ввод данных.

1.5.1. Метод `alert()` – простое модальное окно с сообщением и кнопкой "OK"

Используется для отображения простого окна с сообщением и одной кнопкой - **OK**. Подобное окно может использоваться, например, для отображения сообщений об ошибках (короткий/простой/неправильный пароль). Окно, кроме донесения до пользователя сообщения, больше не предусматривает никакого взаимодействия с пользователем.

Методу `alert()` передается только одна строка - отображаемая строка. Чтобы отобразить многострочное сообщение, разделяйте строки символом \n:

```
window.alert("Привет, мир!");  
window.alert("Привет,\nмир!");
```

Первая наша программа - была не "Привет, мир!", но давайте не будем изменять традиции и таки напишем эту программу, хотя она будет и не первой - хоть тут мы будем отличаться от всех остальных программистов. Ранее мы уже приводили примеры использования данного метода, но рассмотрим сценарий, демонстрирующий его использование, еще раз.

Листинг 1.2. Использование метода `alert()`

```
<html>  
<head>
```

```
<title>Модальное окно с помощью alert</title>
</head>
<body>
<script>
    window.alert("Еще один пример для закрепления
материала!");
</script>
</body>
</html>
```

Наш сценарий находится в теле документа (тег `<body>`), поэтому будет запущен сразу при загрузке HTML-файла. Изображаемое им окно приведено на рис. 1.3.

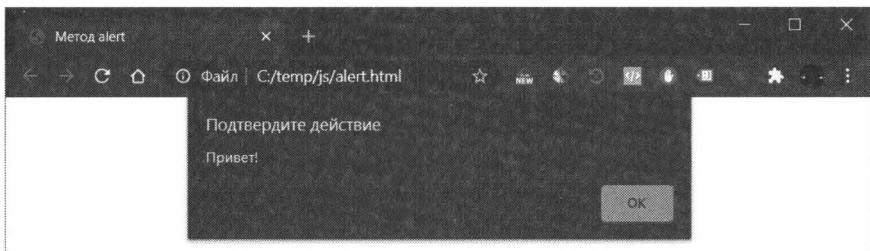


Рис. 1.3. Диалоговое окно в браузере Chrome

1.5.2. Метод `confirm()` - окно с кнопками OK и Cancel

Другой часто используемый метод - метод `confirm()`. Он выводит окно с сообщением и двумя кнопками - **OK** и **Cancel**, позволяя пользователю выбрать одну из кнопок. Проанализировав возвращаемое методом значение (`true`, если нажата кнопка **OK** и `false` - если нажата **Cancel**), вы можете выполнить то или иное действие. Для нашего примера мы будем просто выводить с помощью `console.log()` название нажатой кнопки. Пример использования метода `confirm()` приведен в листинге 1.3. Метод `console.log()` очень полезен для отладки, поскольку позволяет вывести любой текст на консоль браузера, но

при этом пользователи не видят его, что полезно для отладки кода сайтов, которые уже находятся в продакшн.

Листинг 1.3. Использование метода confirm()

```
<html>
<head>
    <title>Диалоговое окно с кнопками OK или Cancel</title>
</head>
<body>
    <script>
        if (window.confirm("Нажмите OK или Отмена")) {
            console.log("Вы нажали OK");
        }
        else {
            console.log("Вы нажали Отмена");
        }
    </script>
</body>
</html>
```

Результат работы этого метода приведен на рис. 1.4.

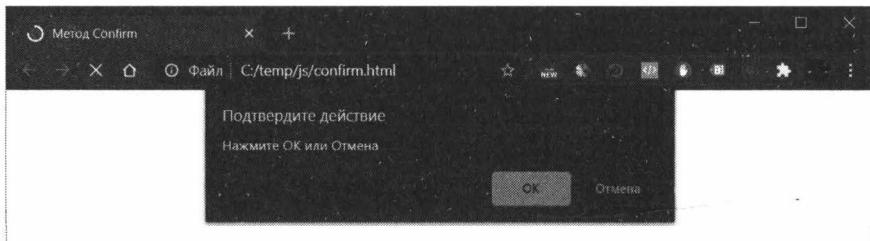


Рис. 1.4. Диалоговое окно, отображаемое методом `confirm` (браузер Chrome)

1.5.3. Метод `prompt()` - диалоговое окно для ввода данных

Позволяет создать диалоговое окно не только с кнопками **OK** и **Cancel** (в русских версиях браузеров – **OK** и **Отмена**), но и с тексто-

вым полем ввода. Введенное в это поле значение можно потом будет присвоить какой-то переменной. Диалог возвращает введенную пользователем строку. Если пользователь ничего не ввел, диалог возвращает значение `null`.

Методу `prompt()` нужно передать два параметра - строку, которая будет отображена в качестве приглашения ввода (над полем для ввода данных) и значение по умолчанию, которое будет передано в сценарий, если пользователь поленится ввести строку и просто нажмет **OK**. Пример использования этого диалога представлен в листинге 1.4.

Листинг 1.4. Пример использования метода `prompt()`

```
<html>
<head>
    <title>Запрос данных от пользователя</title>
</head>
<body>
    <script>
        var UName = window.prompt("Как Вас зовут", "NoName");
        if (UName ==null) {
            alert("пока!");
        }
        else {
            document.write("Привет, " + UName);
        }
    </script>
</body>
</html>
```

Наш сценарий прост. Если пользователь нажмет **Отмена**, то увидит диалог с текстом "пока!" (ну не хотим мы "общаться" с пользователем, который не хочет представиться). Если пользователь нажмет **OK**, то строка из поля ввода будет отображена в HTML-документе в виде "Привет, <введенный текст>".

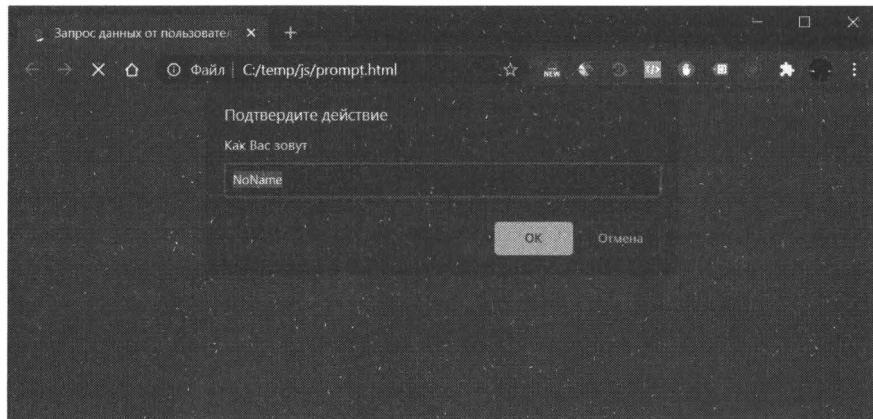


Рис. 1.5. Метод `prompt()` в действии (браузер Chrome)

1.6. Специальные символы

В строках вы можете использовать специальные символы - это комбинации обычных символов, обозначающих служебные или непечатаемые символы, которые нельзя ввести обычным способом. Например, специальный символ `\n` позволяет вставить разрыв строки, что позволяет разбить сообщение в диалоговом окне на строки, например:

```
window.alert("Ошибка! \nПароль слишком короткий");
```

В JS вы можете использовать следующие специальные символы (ради справедливости, нужно отметить, что их можно также использовать в языках C, PHP и некоторых других):

- `\n` – перевод строки;
- `\r` – возврат каретки (в современном программировании используется очень редко);
- `\t` – табуляция;

- \f – перевод страницы;
- ' – апостроф;
- " – двойная кавычка;
- \\ – обратный слеш (косая черта);

1.7. Ключевые слова

Представленные в таблице 1.1 слова нельзя использовать, как идентификаторы, то есть имена переменных и функций.

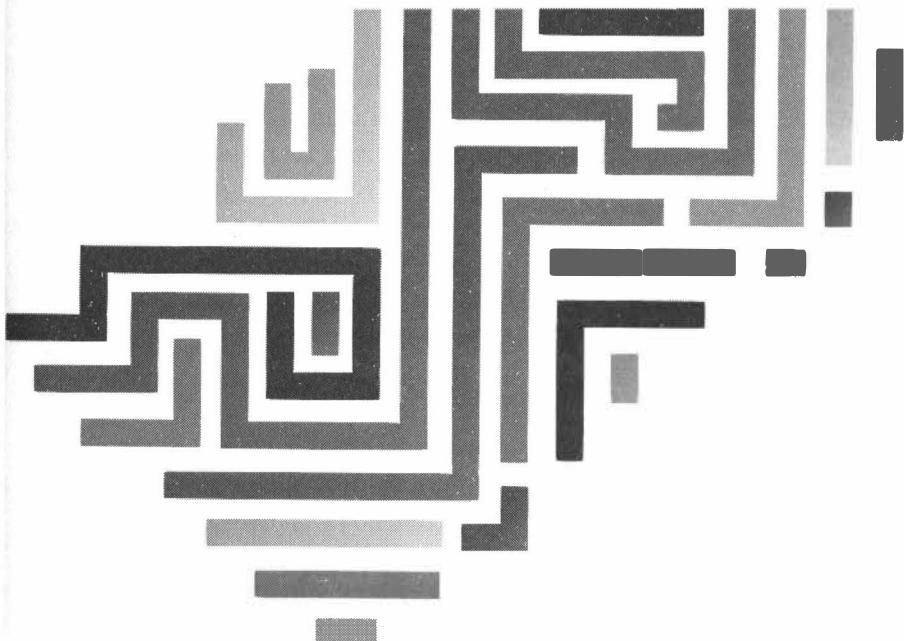
Таблица 1.1. Зарезервированные слова JavaScript

abstract	finally	protected
boolean	float	public
break	for	return
byte	function	short
case	goto	static
catch	if	super
char	implements	switch synchronized
class	import	this
const	in	throw
continue	instanceof	throws
default	int interface	transient
do	long	true
double	native	try
else extends	new	var
false	null	void
final	package	while
	private	with

На этом наша вводная в JavaScript глава заканчивается. В следующей главе мы рассмотрим синтаксис JavaScript. Большая часть материала в этой главе воспринималась интуитивно. Вы все понимали, но все равно у вас есть вопросы относительно синтаксиса языка программирования. Конкретизировать, казалось бы, явные вещи, и предназначена глава 2. В ней мы рассмотрим переменные, операторы и многое другое.

ГЛАВА 2.

Синтаксис JavaScript



2.1. Переменные

2.1.1. Особенности объявления переменных

Переменной называют область памяти, к которой можно обращаться по имени – идентификатору. Во многих языках программирования при объявлении переменных указывается и их типы. Тип переменной определяет тип данных, которые могут храниться в переменной, например, символ, строка, целое число, вещественное число, массив чисел и т.д. В JS при объявлении переменной не указывается тип данных, то есть тип переменной будет зависеть от данных, которые хранятся в ней в данный момент. С одной стороны, это удобно. Сейчас вы присвоили переменной значение 1 и она стала переменной целого типа, а через мгновение вы присваиваете этой же переменной строку символов и тип переменной будет автоматически изменен.

В PHP переменная может быть использована без ее предварительного объявления. В JS переменную нужно сначала объявить с помощью ключевого слова `let` или `var` (далее будет пояснена разница между ними). Желательно при объявлении переменной сразу же ее инициализировать, то есть присваивать начальное значение, например, пустую строку для строковых переменных или 0 для целого числа. В противном случае при первом использовании переменной возникнет ошибка, поскольку значение не определено (`undefined`).

Имя переменной (идентификатор) должно начинаться с символа буквы (A-Z) или символа подчеркивания. Последующими символами могут быть цифры, буквы, а также знак \$. Идентификатор не может начинаться с цифры. Также идентификаторы являются реги-

строзависимыми, то есть переменные `myCar` и `myCar` – это разные переменные.

Имя переменной не может быть зарезервированным словом. Зарезервированные слова JavaScript были приведены в таблице 1.1.

Правильные примеры имен переменных:

`x, y1, userName, _user`

Неправильные имена:

`1x, public`

В первом случае имя переменной начинается с цифры, во втором – является зарезервированным словом.

В JavaScript существует три вида объявлений переменных:

- **var** – объявляет переменную, инициализация переменной значением является необязательной. Область действия переменной – вся функция.
- **let** – объявляет локальную переменную в области видимости блока (то есть кода, заключенного в фигурные скобки), инициализация переменной значением является необязательной.
- **const** – объявляет именованную константу, доступную только для чтения.

Здесь и далее мы будем использовать для объявления переменной ключевое слово **var**. Ключевое слово **let** нужно использовать с осторожностью, иначе можно получить ошибку *ReferenceError: <имя> is not defined*, если обратиться к переменной за пределами блока, в котором она была объявлена. Идеальное использование **let** – в цикле **for** или других подобных инструкциях, например:

```
for (let i = 0; i < prices.length; i++) {  
  ...  
}
```

Как уже было отмечено, переменную настоятельно рекомендуется инициализировать, то есть присвоить ей значение:

```
var my = 100;
```

Можно объявить переменную и без инициализации, однако это нежелательно, поскольку если забыть ее инициализировать перед первым использованием, результаты вычисления могут быть неопределенными (*undefined*).

При желании можно объявить сразу несколько переменных, разделив их запятыми:

```
var x, y1, userName, _user;
```

Сейчас переменная **my** целого типа, поскольку мы ранее присвоили ей значение 100. Но мы можем с легкостью изменить ее тип на строковой, присвоив ей соответствующее значение:

```
my = "hello";
```

Все последующие обращения к переменной производятся без служебного слова **var**.

2.1.2. Типы данных и преобразование типов

Данные, хранящиеся в переменной, могут быть разного типа. Как вы заметили, при объявлении переменной (как это делается в других языках программирования) тип переменной (данных) не указывается.

В JavaScript доступные типы данных можно разделить на две группы: **простые** и **специальные** типы.

К **простым** типам относятся:

- **number** – числа, как целые, так и с плавающей точкой
- **string** – строки

- **Boolean** – логический тип, принимает два значения - *true* (истина) и *false* (ложь)
- **Undefined**
- **Null**
- **Symbol**
- **BigInt**

Специальные типы:

- **массивы**
- **function** – функции. В JS мы можем присвоить ссылку на функцию любой переменной, если указать имя функции без круглых скобок (стоит упомянуть, что в соответствии с ООJS (объектно-ориентированный JavaScript) моделью, функции также представляют собой объекты)
- **object** – массивы, объекты, а также переменные со значением *null*.

Тип переменной интерпретатор JS определяет при присваивании значения, например:

```
Num1 = 5;      // Переменной Num1 присвоено целое значение
               // 5, тип - number
Num2 = 5.5;    // Переменное с плавающей точкой 5.5, тип
               // - number
Str1 = "Hello"; // Переменной Str1 присвоено значение
                 // "Hello", тип - string
Str2 = 'world'; // Также можно использовать одинарные
                 // кавычки
Str3 = null;    // Переменная не содержит данных, ее
                 // тип - object
Booll = true;   // Булевая (логическая) переменная со
                 // значением true
```



Оператор typeof

Тип Num1 - number
Тип Num2 - number
Тип Str1 - string
Тип Str2 - string
Тип Str3 - object
Тип Bool1 - boolean

Рис. 2.1. Определяем тип переменной

С помощью оператора `typeof` мы можем определить тип данных переменной. Он возвращает строку, описывающую тип. Давайте продемонстрируем его работу (см. листинг. 2.1). Сценарий из листинга 2.1 объявляет переменные, выполняет их инициализацию (присваивает значения), а затем выводит тип каждой переменной. Результат работы этого сценария изображен на рис. 2.1.

Листинг 2.1. Оператор `typeof`

```
<html>
<head>
    <title>Оператор typeof</title>
</head>
<body>
<script>
var Num1, Num2, Str1, Str2, Str3, Bool1;

Num1 = 1;
Num2 = 2.5;
Str1 = "Привет";
Str2 = 'Мир';
Str3 = null;
Bool1 = true;

document.write("<br>Тип Num1 - " + typeof(Num1));
document.write("<br>Тип Num2 - " + typeof(Num2));
document.write("<br>Тип Str1 - " + typeof(Str1));
document.write("<br>Тип Str2 - " + typeof(Str2));
document.write("<br>Тип Str3 - " + typeof(Str3));
document.write("<br>Тип Bool1 - " + typeof(Bool1));
```

```
</script>
</body>
</html>
```

Интерпретатор JS "на лету" производит преобразование типов. Подобное преобразование типов аналогичным образом реализовано в других языках программирования. Пусть у нас есть две переменные – одна содержит целое значение, а вторая – строковое:

```
let v1 = 1;
let v2 = "1";
```

Теперь определим еще две переменные:

```
let v3 = v2 + v1;    // результат – строка
let v4 = v1 + v2;    // результат – целое число
```

Тип переменной будет определен по типу первого присваиваемого значения. В нашем случае переменная `v3` будет содержать значение "11", поскольку первой была строковая переменная `v2`. Переменная `v4` будет содержать значение 2.

Переменные в JS также могут быть логического типа (Boolean). Такие переменные могут принимать одно из двух значений – *true* (истина) или *false* (ложь):

```
var Bool = true;
```

При необходимости можно добиться принудительного преобразования типов. Для этого используются следующие функции:

- **parseInt** – преобразует строку в целое число, конечно, если это возможно.
- **parseFloat** – преобразует строку в число с плавающей запятой.
- **eval** – вычисляет выражение в строке, как будто это обычное выражение JS.

Рассмотрим несколько примеров:

```
var A = "1";           // строковое значение "1"
var B = parseInt(A);   // переменная B теперь содержит
                      // число 1
var C = "5.1";         // строка "5.1"
var D = parseFloat(C); // переменной D будет присвоено
                      // число 5.1
var E = "2 + 3";       // строка "2+3"
var F = eval(E);       // переменной F будет присвоено
                      // число 5
```

2.1.3. Области видимости переменной

Очень важной концепцией в JavaScript является область видимости. Область видимости объявляет доступность переменной. Концепция области видимости не такая простая, как можно подумать, и чтобы не запутаться и не совершить ошибок при программировании, нужно со всем разобраться.

Итак, определим переменную:

```
const msg = 'Привет';
console.log(msg);
```

В данном случае мы объявляем переменную и выводим ее на консоль браузера. А теперь мы поместим объявление переменной `msg` в блок `if`, например:

```
if (true) {
    const msg = Привет';
}
console.log(msg); // ReferenceError: message is not
defined
```

При попытке обращения к переменной мы получим ошибку `ReferenceError`. Почему так произошло?

Оказывается, блок `if` создал область видимости для нашей переменной и она доступна только внутри этой области. Поэтому за пределами блока переменная недоступна и мы получаем ошибку.

В других языках программирования есть две области видимости – локальная (когда переменная видима только внутри функции, в которой она объявлена) и глобальная (когда переменная объявлена в программе за пределами функции и доступна везде, в том числе во всех функциях). В JS есть еще видимость блока. Как уже было отмечено, есть три способа определения переменной – **var**, **let** и **const**. Переменная, объявленная с помощью **var**, не имеет блочной видимости. Рассмотрим пример:

```
if (true) {  
    let a = 1;  
    const b = 2;  
    var c = 3;  
}  
console.log(a); // ReferenceError  
console.log(b); // ReferenceError  
console.log(c); // 3
```

Однако переменная, объявленная с помощью **var** – это не глобальная переменная, все зависит от того, где она объявлена. Если она объявлена внутри функции, то она будет доступной только в этой функции:

```
function message() {  
    var msg = 'Привет';  
    console.log(msg);  
}  
message();  
console.log(msg); // ReferenceError
```

В JS есть понятие глобальной области видимости. Она является самой внешней областью. Глобальная область создается при загрузке JS-файла, указанного в атрибуте **src** тега **script**. В этом случае все переменные, объявленные в этом файле за пределами блока и функции, будут глобальными для всего скрипта:

```
<script src="program.js">
```

```
// program.js
// глобальная область видимости
var msg = 'Привет';
```

Область видимости также используется для изоляции переменных. Это позволяет использовать одинаковые имена переменных в разных блоках:

```
function f1() {
    let counter = 1;
    console.log(counter);
}

function f2() {
    let counter = 0;
    console.log(counter);
}

f1();
f2();
```

2.2. Операторы и выражения в JavaScript

2.2.1. Различные типы выражений

Выражением называют набор переменных и операторов. Для каждого выражения вычисляется его результат. Результат может быть строкой, числом или же логическим значением.

Выражение может присваивать значение какой-то переменной, но и есть выражение, результат которых просто вычисляется без присваивания его значения переменной:

```
a = 8 + 8 * 8
100 - 22
```

Существуют еще так называемые условные выражения. Условные выражения определяются так:

Значение1 (условный оператор) Значение2

Используемые в данном формате записи условные операторы называются бинарными, т.к. для их использования необходимо два аргумента (значения).

В последних версиях языка JS появился тернарный условный оператор, использование которого требует трех аргументов. Пример использования тернарного оператора:

переменная = (условие) ? значение1: значение2

Если условие истинно, то переменная примет значение 1, а если нет – значение 2.

2.2.2. Присваивание переменным значений

Операторы присваивания, поддерживаемые в JavaScript, описаны в таблице 2.1.

Таблица 2.1. Операторы присваивания

Оператор	Пример	Описание
<code>+=</code>	<code>a += b</code>	<code>a = a + b</code>
<code>-=</code>	<code>a -= b</code>	<code>a = a - b</code>
<code>*=</code>	<code>a *= b</code>	<code>a = a * b</code>
<code>/=</code>	<code>a /= b</code>	<code>a = a / b</code>
<code>%= (остаток от деления)</code>	<code>a %= b</code>	<code>a = a % b</code>

2.2.3. Арифметические операторы

Математические операторы знакомы всем (они такие же, как в других языках):

- + сложение (например, $A = B + C;$)
- - вычитание (например, $A = B - C;$)
- * умножение (например, $A = B * C;$)
- / деление (например, $A = B / C;$)
- % деление по модулю (например, $A = B \% C$)
- ++ инкремент, увеличивает значение переменной на 1 (например, $i++;$)
- -- декремент, уменьшает значение переменной на 1 (например, $j--;$)

Как использовать большинство из этих операторов, понятно даже новичкам. Нужно отметить разве что особенность операторов инкремента и декремента. Пример:

```
a = 1000  
b = a++
```

Переменной **b** будет присвоено значение 1000, а после этого переменная **a** будет увеличена на 1 (значение 1001). Если же **++** указать до **a** (а не после него), то сначала переменная **a** будет увеличена на 1, а потом уже будет присвоено новое значение переменной **b**:

```
b = ++a
```

2.2.4. Булевые операторы

К булевым операторам относятся следующие:

- ! – унарная операция отрицания (NOT)

- `&&` – бинарная операция И (AND): результат `true`, когда оба операнда истинны
- `||` – бинарная операция ИЛИ (OR): результат `true`, когда один из operandов равен `true`

Пример:

```
bool = true && false
```

2.2.5. Операторы сравнения

Таблица 2.2 содержит операторы сравнения, которые вы можете использовать в JavaScript при написании своих сценариев.

Таблица 2.2. Операторы сравнения в JavaScript

Оператор	Назначение
<code>></code>	Больше
<code>>=</code>	Больше или равно
<code><</code>	Меньше
<code><=</code>	Меньше или равно
<code>==</code>	Равно
<code>!=</code>	Не равно
<code>====</code>	Строго равно

Обратите внимание, что в JS есть два оператора сравнения: равно (`==`) и строго равно (`====`). В чем между ними разница? Оператор `==`, сравнивая значения разных типов, пробует свести типы, а затем выполнить сравнения. То есть значение `3` будет равно строке `"3"`. Оператор `====`, встретив разные типы значений, сразу вернет `false`.

2.2.6. Двоичные операторы

К бинарным (двоичным) операторам относятся следующие операторы:

- \sim двоичная инверсия ($X = \sim Y;$);
- $\&$ двоичное И ($X = Y \& Z;$);
- $|$ двоичное ИЛИ ($X = Y | Z;$);
- \wedge двоичное исключающее ИЛИ ($X = Y \wedge Z;$);
- $<<$ сдвиг влево на один или более разрядов с заполнением младших разрядов нулями ($Z = X << Y;$);
- $>>$ сдвиг вправо на один или разрядов с заполнением старших разрядов содержимым самого старшего разряда ($X = Y >> Z;$)
- $>>>$ сдвиг вправо без учета знака, старшие разряды будут заполнены нулями ($X = Y >>> Z.$).

Бинарные операторы предназначены для выполнения поразрядных операций с двоичным представлением целых чисел.

2.2.7. Конкатенация

Для слияния (конкатенации) строк используется оператор `+`:

```
var test = "hello " + "world"
```

Переменная `test` в результате будет содержать значение "hello world".

2.2.8. Приоритет выполнения операторов

Сейчас мы поговорим о приоритете выполнения операторов. Пусть у нас есть выражение:

```
A = 2 + 3 * 4 / 5;
```

Последовательность вычисления значения этого оператора известна даже школьникам: сначала **3** будет умножено на **4** (в результате мы получим значение **21**) – ведь приоритет операции умножения выше, чем сложения. После этого значение **21** будет разделено на **5** и мы получим значение **4.2** (приоритет операции деления выше, чем сложения).

Далее к значению **4.2** будет добавлено значение **2** и в результате переменной **A** будет присвоено значение **6.2**.

С помощью скобок можно изменить приоритет выполнения операций:

```
A = (2 + 3) * 4 / 5;
```

В этом случае сначала будет вычислено выражение **2 + 3**, а затем полученное значение будет умножено на **4** и разделено на **5**. Приоритет операций умножения и деления одинаковый, поэтому они выполняются слева направо, то есть *в порядке следования*. В результате будет получено значение **4** (**5 * 4** и разделить на **5**).

Операторы в порядке убывания приоритета:

- **!, ~, ++, --** отрицание, двоичная инверсия, инкремент, декремент;
- ***, /, %** умножение, деление, остаток от деления;
- **+, -** сложение и вычитание;
- **<<, >>, >>>** двоичные сдвиги;
- **&** двоичное И;
- **^** двоичное исключающее ИЛИ;
- **|** двоичное ИЛИ;
- **=, +=, -=, *=, /=, %=** присваивание.

2.3. Основные конструкции языка

К основным конструкциям языка относят условный оператор (`if..else`), а также операторы циклов. В этом разделе будут рассмотрены эти конструкции.

2.3.1. Условный оператор `if`

Прежде, чем мы будем рассматривать условный оператор `if`, настоятельно рекомендую вернуться к разделу 2.2.5 и еще раз просмотреть таблицу с операторами сравнения - так вам будет понятнее все происходящее здесь.

Условный оператор `if` позволяет выполнить определенное действие в зависимости от истинности условия. Общая форма оператора выглядит так:

```
if (условие)
    {операторы, если условие истинно}
[else {
    операторы, если условие ложно}]
```

Обратите внимание, что вторая часть (`else`) не обязательна.

Условие - это логическое выражение, построенное на базе операторов сравнения, именно поэтому я просил вас вернуться к разделу 2.2.5, чтобы еще раз просмотреть имеющиеся операторы сравнения. Каждый из операторов сравнения возвращает `true` в случае истинности и `false`, если проверяемый факт ложен.

Пусть у нас есть две переменные:

```
var A = 10;
var B = 5;
```

Оператор `A == B` вернет *false*, поскольку `A` не равно `B`. Оператор `A > B` вернет *true*, поскольку `A` больше, чем `B`.

Чтобы инвертировать булевое значение, вы можете использовать оператор `!`, например:

```
!(X == Z)
```

Можно также использовать оператор `!=`, но используйте ту форму оператора, которая вам удобнее.

В нашем случае переменные `X` и `Z` не равны, поэтому оператор `==` вернет значение *false*, но поскольку указан оператор `!`, то будетозвращено значение *true*.

Рассмотрим несколько примеров:

```
var A = 10;
var B = 5;

// Будет выведено A > B
if (A > B) {
    document.write('A > B'); }

// Будет выведено B < A
if (A > B) {
    document.write('A > B'); }
else {
    document.write('B < A'); }

// Будет выведено A = B
if (!(A == B)) {
    document.write(" A != B "); }
else {
    document.write(" A = B "); }
```

В главе 1 мы уже сталкивались с оператором `if` и рассмотрели более сложный пример - проверку нажатия одной из кнопок в диалоговом

окне. Вы можете вернуться к тому примеру, чтобы освежить его в памяти.

Операторы **if** можно вкладывать друг в друга, что продемонстрировано в листинге 2.2. Сценарий в этом листинге пытается разделить 10 на значение одной из переменных - **A** или **B**, предварительно проверяя, не равно ли значение этих переменной 0. Сначала сценарий проверяет, не равно ли 0 значение переменной **A**. Поскольку **A** = 0, то выполнение сценария переходит на второй оператор **if**, который проверяет, не равно ли 0 значение переменной **B**. С переменной **B** все хорошо, поэтому переменной **C** будет присвоено значение $10 / 1$ (10). В противном случае, если обе переменные равны 0, будет выведено сообщение *Деление на 0*.

Листинг 2.2. Вложенность операторов if

```
<html>
<head>
    <title>Пример использования оператора if</title>
</head>
<body>
<script>

var X = 0;
var Y = 1;
var Z;

if (X !=0) {
    Z = 10 / X;
    document.write('Z = ' + Z); }
else if (Y != 0) {
    Z = 10 / Y;
    document.write('Z = ' + Z); }
else document.write("Ошибка! Деление на 0");

</script>
</body>
</html>
```

2.3.2. Оператор выбора switch

Иногда (конечно не во всех ситуациях) вместо множества вложенных операторов **if** можно использовать оператор **switch**. Оператор **switch** позволяет сравнить переменное или выражение с множеством значений, что позволяет избавиться от серии операторов **if** и сделать код более компактным.

Общая форма оператора **switch** выглядит так:

```
switch (<Переменная или выражение>) {  
    case <Значение 1>:  
        <Оператор 1>;  
        break;  
    case <Значение 2>:  
        <Оператор 2>;  
        break;  
    ...  
    [default:  
        <Оператор>]  
}
```

Работает данный оператор следующим образом:

- Вычисляется значение переменной или выражения;
- Полученное значение сравнивается с одним из значений, указанных в блоках **case**;
- Пусть у нас 10 блоков **case**, и значение совпало с 5-ым блоком **case**. Тогда, если в 5-ым блоке **case** не указан оператор **break**, то будут выполнены действия, связанные с блоками 5-10, а также операторы из блока **default**. Если же указан оператор **break**, тогда будет выполнено только то действие, которое указано в 5-ом блоке **case**. Для большей однозначности (если не нужно иного) я всегда рекомендую использовать оператор **break** для преждевременного выхода из оператора **switch**;

- Если полученное значение не совпало ни с одним из значений, указанных в блоках **case**, тогда будут выполнены операторы из блока **default**, если таковой указан. Блок **default** является необязательным.

Представим, что у нас есть переменная **command**, в зависимости от значения которой нужно выполнить определенные действия, например:

```
if (command == 1) alert('Выбрано действие: 1');
if (command == 2) alert('Выбрано действие: 2');
if (command == 3) alert('Выбрано действие: 3');
if (command == 4) alert('Выбрано действие: 4');
```

Код выглядит громоздко и логически не воспринимается, как один блок, а как четыре разных блока (если бы мы по этому коду построили блок-схему, то у нас бы и получилось четыре разных блока).

Весь этот громоздкий код мы можем заменить на более компактный. Пусть он занимает больше строк, зато выглядит не таким перегруженным и воспринимается как единое целое:

```
switch (command) {
    case 1: alert('Выбрано действие: 1'); break;
    case 2: alert('Выбрано действие: 2'); break;
    case 3: alert('Выбрано действие: 3'); break;
    case 4: alert('Выбрано действие: 4'); break;
    default: alert('Неизвестное действие!');
}
```

Как видите, получившийся код воспринимается не таким перегруженным, хотя занимает больше строк. К тому же оператор **switch** позволяет задавать действие по умолчанию. В конечном итоге, с его помощью можно довольно понятнее и прозрачнее реализовывать сложные разветвления, которые кажутся запутанными, если их реализовать с помощью **if**. Однако еще раз отмечу, что **switch** - далеко не панацея во всех ситуациях выбора.

В листинге 2.3 приведен пример использования оператора **switch**. Сначала мы отображаем диалог ввода действия, затем анализируем, какое действие выбрал пользователь. Обратите внимание, что прежде, чем передать полученное действие оператору **switch**, мы сводим его к типу **number** с помощью функции `parseInt()`.

Листинг 2.3. Пример использования оператора **switch**

```
<html>
<head>
    <title>Пример использования switch</title>
</head>
<body>
    <script>
        var command = window.prompt("Введите действие", "");

        if (command == null) {
            document.write('Нажата кнопка Отмена');
        }
        else
            switch (parseInt(command)) {
                case 1: alert('Выбрано действие 1'); break;
                case 2: alert('Выбрано действие 2'); break;
                case 3: alert('Выбрано действие 3'); break;
                case 4: alert('Выбрано действие 4'); break;
                default: alert('Неизвестное');
            }
    </script>
</body>
</html>
```

2.3.3. Циклы

Если проанализировать все программы, то на втором месте после условного оператора будут операторы цикла. Используя цикл, вы можете повторить операторы, находящиеся в теле цикла. Количество

повторов зависит от типа цикла - можно даже создать бесконечный цикл. В JavaScript есть три типа цикла:

- Цикл **for** или цикл со счетчиком
- Цикл **while** или цикл с предусловием
- Цикл **do..while** или цикл с постусловием

Цикл со счетчиком

Данный цикл используется для выполнения тела цикла четко определенного количества раз. Цикл **while**, например, удобно использовать для ожидания какого-то события (мы не знаем, сколько раз будет выполнено тело цикла, пока условие станет истинным), а цикл **for** используется тогда, когда вы точно знаете, сколько раз нужно повторить цикл. Синтаксис цикла **for**:

```
for (<команды_инициализации>; <условие>;
      <послеитерационные_команды>) {
    <Тело цикла>
}
```

Первым делом выполняются команды инициализации. Команды инициализации выполняются всего лишь один раз. Далее цикл **for** проверяет условие: если оно истинно, выполняется тело цикла. После того, как будет выполнен последний оператор тела, выполняются команды "после итерации". Затем цикл **for** опять проверяется условие, в случае, если оно истинно, выполняется тело цикла и поститерационные команды и т.д.

Выведем строку 0123456789:

```
for (i=0; i<10; i++) document.write(i);
```

Чтобы вывести строку 12345678910 нужно установить начальное значение счетчика в 1 и изменить условие цикла:

```
for (i=1; i<=10; i++) document.write(i);
```

Цикл **for** будет очень полезным при обработке массивов, которые мы рассмотрим в следующей главе.

Цикл **while**

В некоторой мере цикл **for** очень похож на цикл с предусловием (**while**), так как сначала проверяется условие, а потом уже выполняется тело цикла. Рассмотрим цикл с предусловием:

```
while ( логическое выражение ) {  
    операторы; }
```

Сначала цикл вычисляет значение логического выражения. Если оно истинно, выполняется тело цикла, иначе происходит выход из цикла и выполнение следующего за циклом оператора.

Рассмотрим пример, выводящий числа от 1 до 10:

```
var i = 1;  
  
while (i < 11) {  
    document.write(i + "<br>");  
    i++;  
}
```

Соблюдайте осторожность при использовании цикла **while**. Если вы не предусмотрите условие выхода, тогда вы получите бесконечный цикл. В нашем случае условием выхода является не только, что $i < 11$, но и сам инкремент **i**, то есть оператор, способный повлиять на условие, указанное в заголовке цикла. Если бы мы забыли указать оператор **i++**, то мы бы получили бесконечный цикл. Поскольку переменная **i = 1**, что меньше 11 и у нас нет другого оператора, который

в теле цикла изменял бы эту переменную, мы тело цикла будет выполнять бесконечно.

Цикл do..while

В отличие от цикла **while**, сначала выполняются операторы (тело цикла), а затем уже проверяется условие. Если условие истинно, то начинается следующая итерация. В случае с циклом **do..while** тело цикла будет выполнено как минимум один раз. Синтаксис цикла:

```
do
{
    // тело цикла
}
while (условие);
```

Пример:

```
i = 1;
do {
    document.write(i);
    i++;
} while (i < 10);
```

Управление выполнением цикла. Операторы **break** и **continue**

В теле любого цикла разрешается использовать операторы **break** и **continue**. Оператор **break** прерывает выполнение цикла, а оператор **continue** – прерывает выполнение текущей итерации и переходит к следующей.

Пусть нам нужно вывести только нечетные числа в диапазоне от 1 до 20. Пример цикла может быть таким:

```
for (i=1; i<21; i++) {  
    if (i % 2 == 0) continue;  
    else document.write(i + " ");  
}
```

В теле цикла мы проверяем, если остаток от деления **i** на 2 равен 0, значит, число четное и нужно перейти на следующую итерацию цикла. В противном случае нам нужно вывести наше число.

А вот пример использования оператора **break**. Не смотря на то, что цикл якобы должен выполниться 10 раз, он будет прерван, когда **i** будет равно 5:

```
for (i=1; i<11; i++) {  
    if (i == 5) break;  
    document.write(i + " ");  
}
```

В результате будет выведена строка:

1 2 3 4

Вложенность циклов

В теле цикла может быть другой цикл. Вложенность циклов формально не ограничивается, однако нужно быть предельно осторожным, чтобы не допустить зацикливания. Пример вложенного цикла:

```
var j = 1;  
while (j < 15) {  
    for (k=0; k<j; k++) document.write('*');  
    document.write('<br>');  
    j++;  
}
```

Результат выполнения этого сценария приведен на рис. 2.2.

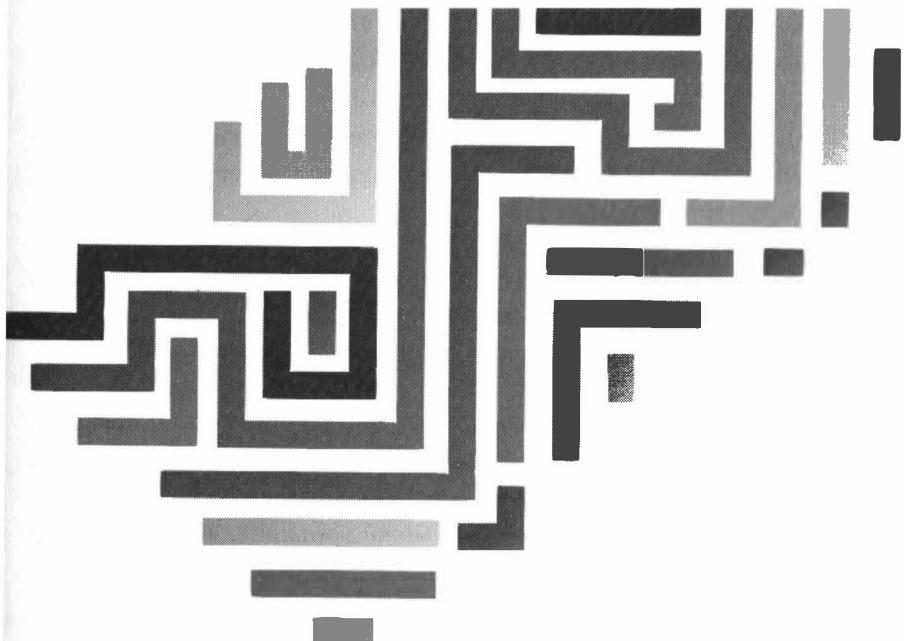
Одно из частых применений циклов - обработка массивов. О том, что такое массивы и как с ними работать в JavaScript, мы поговорим в следующей главе.

*
**

Рис. 2.2. Пример работы вложенных циклов

ГЛАВА 3.

Массивы в JavaScript



3.1. Введение в массивы

Все мы знаем, что такая переменная - это поименованная область в памяти. Мы выделяем область в памяти (точнее за нас это делает интерпретатор) и назначаем ей имя (а вот имя выбираем мы - программисты). Но переменные не всегда удобны. Представим, что нам нужно хранить набор данных, состоящий из 100 или даже 1000 значений одного типа. В этом случае нам понадобится 100 или 1000 переменных, что не совсем удобно. Специально для таких целей были созданы массивы. Массив - это упорядоченный набор данных. Каждый элемент массива обладает своим индексом (также индекс называют ключом), который однозначно идентифицирует элемент внутри массива. В массиве не может быть двух или более элементов с одинаковым значением индекса.

Работать с массивами намного удобнее, поскольку можно в цикле перебрать все элементы массива, а не обращаться отдельно к каждому элементу (как это происходит с переменными).

Если вы программируете на других языках программирования, то знаете, что все переменные, в том числе и массивы, должны быть объявлены в строго определенном месте программы. В JavaScript переменную, следовательно, и массив, можно объявить в любом месте программы (сценария), но до первого использования массива. А это удобно, если нужно создать массив с неизвестным числом параметров. Например, вам нужно прочитать в массив текстовый файл так, чтобы каждая строка файла представляла собой один элемент массива.

ва. В другом языке программирования нужно было объявить очень большой массив, скажем, на 10 000 элементов - ведь вы же не знаете, сколько строк будет в файле, но нужно написать более или менее универсальную программу, чтобы она умела обрабатывать большие файлы. А что делать, если в файле всего 3 строки или 10 001 строка? В первом случае вы выделили памяти во много раз больше, чем реально нужно для обработки файла. Ваша программа будет попросту поедать память, и вы ничего не сможете сделать. Во втором случае ваша программа не справится с обработкой всего файла, поскольку в файле больше строк, чем может поместиться в массив.

3.2. Инициализация массива

Массив инициализируется, как и любая другая переменная - путем присваивания значения. Поскольку массив может содержать несколько значений, при его инициализации значения указываются в квадратных скобках и разделяются запятыми:

```
var M;  
M = [5, 3, 4, 1, 2];  
  
var Months;  
Months = ["", "Янв", "Фев", "Мар", "Апр", "Май",  
"Июн", "Июл", "Авг", "Сен", "Окт", "Ноя", "Дек"];
```

Нумерация элементов массива начинается с 0, а не с 1. Получить доступ к определенному элементу массива можно так:

```
num0 = M[0]; // 5  
j = M[3]; // 1
```

3.3. Изменение и добавление элементов массива

При желании вы всегда можете изменить значение существующего элемента массива или добавить новый элемент, например:

```
M[0] = 7;  
M[5] = 8;
```

Первый оператор присваивает значение 7 элементу массива M с индексом 0. Второй оператор создает новый элемент массива со значением 8, индекс элемента - 5. Теперь у нас есть вот такой массив:

```
[7, 3, 4, 1, 2, 8]
```

Посмотрим, что сделает следующий оператор:

```
M[7] = 11;
```

Этот оператор создаст два элемента массива: элемент с индексом 6 и значением *undefined* и элемент с индексом 7 и значением 11. Массив будет следующим:

```
[7, 3, 4, 1, 2, 8, undefined, 11]
```

3.4. Многомерные массивы

В JS вы можете создавать многомерные массивы путем присваивания любому элементу массива нового массива, например:

```
M[0] = [3, 2, 1];
```

Обратиться к элементу многомерного массива можно так:

```
j = M[0][1];
```

3.5. Пример обработки массива

Подробно о средствах работы с массивами мы поговорим позднее, когда мы будем рассматривать встроенные классы JS. Сейчас же мы рассмотрим несколько примеров для работы с массивами. У любого объекта массива есть свойство **length**, содержащее длину массива. Это свойство мы можем использовать при обработке массива.

Сценарий из листинга 3.1 вычисляет минимальный элемент массива. Сначала минимальным считается первый (с индексом 0) элемент массива. Далее в цикле **for** мы сравниваем каждый следующий элемент массива. Если сравниваемый элемент меньше нашего минимума, он становится новым минимумом.

Листинг 3.1. Вычисление минимума массива

```
<html>
<head>
    <title>Поиск минимума в массиве</title>
</head>
<body>
    <script>
        var M = [9, 11, 7, 4, 99, 3, 2, 33, 3];

        Min = M[0];      // Минимум .
        Min_ind = 0;    // Индекс минимума

        document.write('Массив: <br>' + Min + " ");

        for (i=1; i<M.length; i++) {
            if (M[i] < Min) {
                Min = M[i];
                Min_ind = i;
            } // if
            document.write(M[i] + " ");
        } //for
```

```

document.write('<br>Минимум: ' + Min);
document.write('<br>Индекс: ' + Min_ind);

</script>
</body>
</html>

```

Сначала мы считаем минимальным элемент с индексом 0. Далее выводим этот минимальный элемент, поскольку обработка массива начинается с индекса 1 (сравнивать 0-ой элемент с 0-ым элементом нет смысла). В цикле мы проверяем, не является ли текущий элемент минимальным и, если это так, устанавливаем новый минимум. Для идентификации минимума используются две переменных - `Min` (минимум) и `Min_ind` (индекс минимума). Также в цикле мы выводим обрабатываемый элемент. Результат работы сценария приведен на рис. 3.1.



Рис. 3.1. Поиск минимального элемента

Теперь мы решим другую подобную задачу - найдем максимальный элемент. Чтобы было интереснее, давайте сделаем двумерный массив и найдем в нем максимальный элемент. А то бы наш новый сценарий отличался бы только знаком больше от предыдущего. Согласитесь, совсем не интересно. В новом же сценарии нам нужно обойти два измерения. Для обхода первого измерения мы будем использовать счетчик `i`, для обхода второго - `j`. Как только найдем максимальный

элемент, в массив **Ind** мы запишем его "координаты" - значения счетчиков **i** (элемент 0) и **j** (элемент 1).

Листинг 3.2. Поиск максимума в двухмерном массиве

```
<html>
<head>
    <title>Максимум в массиве</title>
</head>
<body>
    <script>
        var M = [0, 0, 0];

        M[0] = [3, 2, 1];
        M[1] = [7, 8, 9];
        M[2] = [5, 6, 7];

        Max = M[0][0];      // Максимум
        Ind = [0, 0];       // Индекс максимума

        for (i=0; i<M.length; i++) {
            for (j=0; j<M[i].length; j++) {

                if (M[i][j] > Max) {
                    Max = M[i][j];
                    Ind[0] = i;
                    Ind[1] = j;
                }
            }
        }

        document.write('Максимум ' + Max);
        document.write('<br>Ind [' + Ind[0] + '][' + Ind[1] +
        ']');
    </script>
</body>
</html>
```

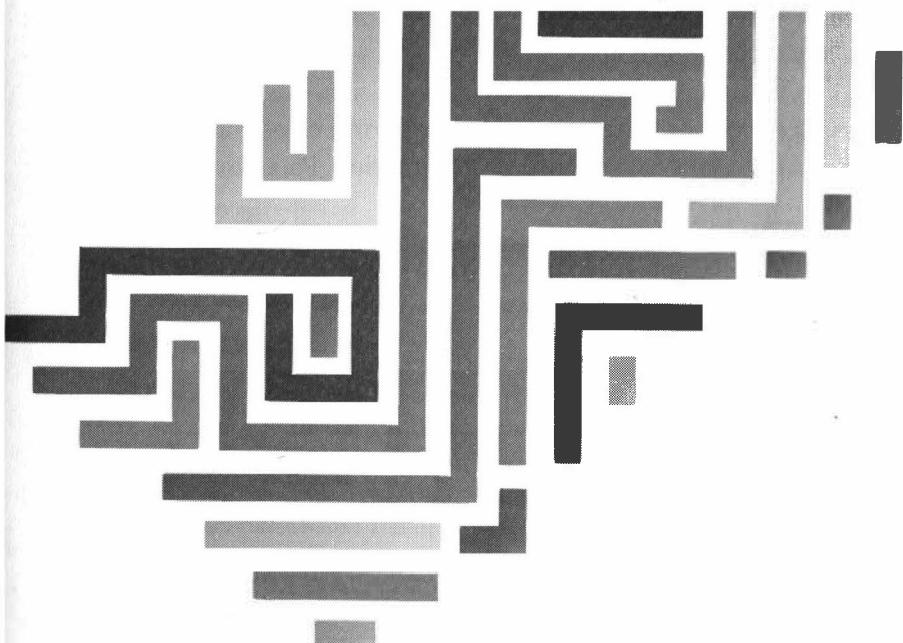
Обратите внимание, как мы обращаемся к свойству **length**. Поскольку каждый из элементов исходного массива также является массивом, то мы можем обратиться к **length** так: `M[i].length`. Конечно, мы и так знаем размерность массива, и можно было бы использовать значения длины 3 для каждого из счетчиков, но корректнее. Результат работы сценария 3.2 изображен на рис. 3.2.



Рис. 3.2. Поиск максимума в двухмерном массиве

ГЛАВА 4.

Функции в JavaScript



4.1. Основные понятия

4.1.1. Способы объявления функций

Функция - это фрагмент JavaScript-кода. По сути, функция - это подпрограмма.

Существует три способа объявления функции. Первый, наиболее универсальный способ, в англоязычной литературе называется как *function declaration statement*. Он заключается в использовании ключевого слова **function** и подходит для объявления любых функций – от самых простых до самых сложных. В этом случае функция описывается с помощью ключевого слова **function**. Рассмотрим синтаксис:

```
function <Имя> ([<Параметры функции>]) {  
    <Тело функции - операторы>  
    [return <Возвращаемое значение>]  
}
```

Имя функции – это тоже идентификатор, поэтому для имени функции используются те же правила, что и для имени переменной. Поэтому имя функции должно быть уникальным внутри скрипта. Параметры функции разделяются запятыми, если параметров нет, то указываются только круглые скобки.

В фигурных скобках описывается тело функции - выражения JavaScript. Обычно эти выражения производят обработку переданных функции параметров.

Функция должна возвращать результат. Для возвращения результата предусмотрено ключевое слово **return**. Иногда функция может не возвращать никакого значения. Например, функция может выводить сообщение в блок **div**. В этом случае она может не возвращать никакого значения – возвращать то нечего.

Второй способ называется функция-выражение (*function definition expression*). Он также подразумевает использование ключевого слова **function**, но при этом не задается имя самой функции. В других языках программирования такие функции называются анонимными. Пример объявления функции:

```
var sum = function(a, b) { return a+b; };
var x = sum(2, 2);
```

Такие функции удобно использовать, когда функция передается аргументов в другую функцию. Рассмотрим небольшой пример:

```
function map(f, a) {
    var resFunc = []; // Создаем новый массив
    var i; // Объявляем переменную
    for (i = 0; i != a.length; i++)
        resFunc[i] = f(a[i]);
    return resFunc;
}
var f = function(z) {
    return z * z * z;
}
var nums = [0, 5, 7, 9, 12];
var cube = map(f,nums);
console.log(cube);
```

В консоли разработчика мы получим следующие значения [0, 125, 343, 729, 1728].

Третий способ создания функции – функции-стрелки или стрелочные функции. Данный способ подходит для простых функций функций, и он предпочтительнее, чем использование универсального способа из-за своей компактности. Синтаксис выглядит так:

```
let func = (arg1, arg2, ...argN) => expression
```

Небольшой пример:

```
let sum = (a, b) => a + b;  
console.log(sum(2, 2));
```

4.1.2. Практические примеры

Рассмотрим несколько примеров функций:

```
// Функция просто выводит диалоговое окно с текстом 'Access denied'  
// Использование этой функции просто короче, чем вызов windows.alert  
// К тому же, когда понадобится изменить текст сообщения, тогда  
// текст можно будет изменить в одном месте, а не по всему сценарию  
// Функция ничего не возвращает  
function denied() {  
    window.alert('Access Denied!');  
}  
  
// Функция возвращает сумму двух чисел. Никакой проверки, являются  
// ли аргументы числами, не производится  
function Sum(x, y) {  
    var result = x + y;  
    return result;  
}
```

Использовать эти функции можно так:

```
denied();           // будет выведено наше сообщение  
var x = Sum(2, 2); // в переменную x будет записан  
                   // результат функции Sum
```

Как только выполнен оператор **return**, функция завершает работу. Все операторы после оператора **return** не будут выполнены, например:

```
function Sum(x, y) {  
    var result = x + y;  
    return result;  
    window.alert('Sum'); // никогда не будет выполнен  
}  
//
```

В операторе **return** можно указать сразу выражение, перепишем нашу функцию:

```
function Sum(x, y) {  
    return (x + y);  
}
```

Функции можно передавать не только значения, но и значения переменных, например:

```
var a = 10;  
var b = 12;  
  
var ab = Sum(a, b);
```

Ссылка на функцию может быть сохранена в любой переменной, например:

```
var d = denied; // Имя функции указываем без скобок  
d(); // Вызываем функцию denied() по ссылке
```

В JS допускаются также анонимные функции (функции без названия):

```
var f = function() { // ссылка на анонимную функцию  
    window.alert('Привет'); // присваивается переменной x  
}  
f(); // вызываем функцию через переменную x
```

Ссылка на вложенную функцию может быть возвращена в качестве значения конструкции **return**, для этого используются круглые скобки два раза. Пример:

```
var x = function() { // Ссылка на анонимную функцию
    return function() { // Возвращаем ссылку на вложенную
        // функцию
        window.alert("Привет");
    };
};

x()(); // Вызываем вложенную функцию
```

4.2. Размещение функций внутри сценария

Мы уже рассмотрели достаточно примеров функций, осталось только понять, где в HTML-документе должны находиться функции. Теоретически, функция может находиться в любом месте сценария, но до первого момента ее использования. Чтобы не запутываться, прежде всего, самих себя, программисты обычно помещают описание функций в отдельный JS-файл и подключают его в HTML-коде до первого вызова этих функций. Обычно так поступают, если функций достаточно много или код функций слишком объемный. Второй способ, который на данный момент не рекомендуется – объявление функций в секции HEAD документа.

Второй способ, который на данный момент не рекомендуется – объявление функций в секции HEAD документа. В современных реалиях все скрипты, в том числе и описание новых функций, рекомендуется заключать в конце тега BODY. Важно, что функции должны быть объявлены до их первого использования. Сейчас мы рассмотрим оба варианта.

Листинг 4.1. Функция помещена в HEAD

```
<html>
<head>
    <title>Функции</title>
    <script>
        function denied() {
            window.alert('Access Denied!');
        }

    </script>
</head>
<body>
    <script>

        denied();      // вызываем функцию

    </script>
</body>
</html>
```

В листинге 4.2 мы подключаем JS-файл functions.js (имя файла может быть любым). Код файла functions.js приведен в листинге 4.3.

Примечание. Для современных браузеров, совместимых со стандартом HTML5, атрибут type = "text/javascript" указывать не нужно!

Листинг 4.2. Вызов функции из внешнего JS-файла

```
<html>
<head>
    <title>Функции</title>
    <script type="text/javascript" src="libraryfunctions.
js"></script>
</head>
<body>
    <script>
```

```
message();  
  
</script>  
</body>  
</html>
```

Листинг 4.3. Внешний JS-файл (libraryfunctions.js)

```
function message() {  
    window.alert('Access Denied!');  
}
```

Понятно, не нужно создавать отдельный JS-файл для каждой функции. Вы можете создать один-единственный файл, в который вы поместите все функции, которые необходимы вашему основному сценарию.

4.3. Рекурсия

Рекурсия - явление, когда функция вызывает саму себя. Нужно отметить, что рекурсивные алгоритмы очень опасны и их рекомендуется по возможности избегать. Основная опасность в зацикливании, когда не предусмотрено (или предусмотрено некорректное) условие выхода из рекурсии. Во многих книгах по программированию рекурсия традиционно используется для вычисления факториала. Далее приведена функция Factorial(), вычисляющая факториал числа x. Условием выхода из рекурсии является оператор:

```
if (x == 0 || x == 1) return 1;
```

Если x равен 0 или 1, функция вернет 1, в противном случае функция будет вычислять факториал x - 1, для чего вызовет саму себя. Код функции:

```
function f_Factorial(x) {
```

```
if (x == 0 || x == 1) return 1;
else return (x * f_Factorial(x - 1));
}
```

Вы должны знать, что любой рекурсивный алгоритм можно преобразовать в нерекурсивный. По возможности старайтесь избегать рекурсии.

4.4. Область видимости при использовании функций

Глобальными по отношению к функции являются все переменные, объявленные за пределами функции. Такие переменные доступны в любой части скрипта, в том числе и в функции.

Локальными считаются переменные, которые были объявлены в самой функции. Локальные переменные доступны только внутри функции, в которой они объявлены и недоступны в других функциях или в основной программе.

Если имя локальной переменной совпадает с именем глобальной переменной, то будет использоваться локальная переменная, а значение глобальной переменной останется без изменения.

Рассмотрим листинг 4.4. В секции HEAD мы объявляем две переменные - **A** и **B**. Они будут глобальными переменными, доступными, как в функции F1(), так и в коде из секции BODY, то есть везде по сценарию. В теле функции мы объявляем две локальные переменные - **X** и **B**. Затем функция выводит значение переменных A, B, X. Посмотрите на рис. 4.1. Функция вывела значения 10, 5 и 10. Как видите, используется локальная переменная **B** вместо глобальной переменной **B**, если имена переменных совпадают. Основная программа выводит значения переменных **A** и **B**, будут выведены значения 10 и 20. Выводить значение переменной **X** в основной програм-

ме нет смысла, так как вы получите сообщение об ошибке *Uncaught ReferenceError: X is not defined.*

Листинг 4.4. Глобальные и локальные переменные

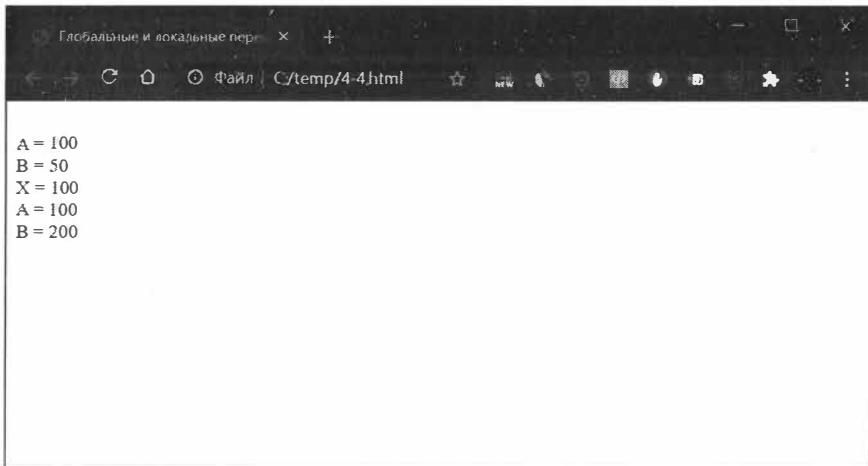
```
<html>
<head>
    <title>Глобальные и локальные переменные</title>
    <script>
        // глобальные переменные
        var A = 100;
        var B = 200;

        function F1() {
            // локальные переменные функции
            var X = 100;
            var B = 50;

            document.write("<br>A = " + A);
            document.write("<br>B = " + B);
            document.write("<br>X = " + X);

        }
        </script>
    </head>
    <body>
        <script>
            F1();
            document.write("<BR>");
            document.write("A = " + A);
            document.write("<br>B = " + B);

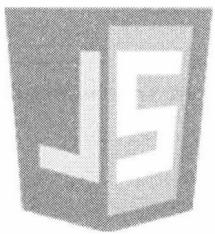
        </script>
    </body>
</html>
```



The screenshot shows a browser window titled "Глобальные и локальные пер." (Global and local variables). The address bar indicates the file is C:/temp/4-4.html. The page content displays the following output from a script:

```
A = 100
B = 50
X = 100
A = 100
B = 200
```

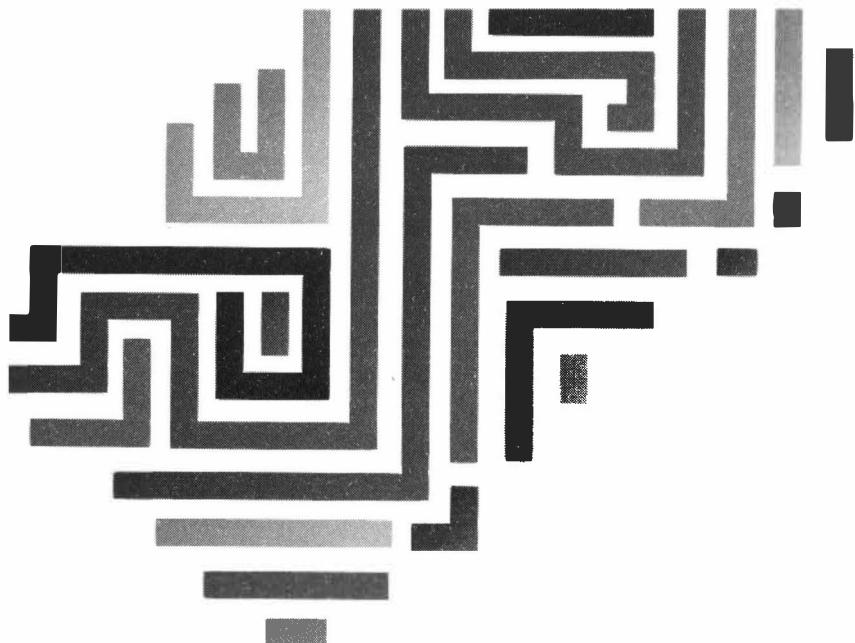
Рис. 4.1. Область видимости локальных и глобальных переменных



JavaScript

ГЛАВА 5.

Отладка программ и обработка ошибок



Прежде, чем наши сценарии станут очень сложными, нужно поговорить об отладке сценариев и обработке ошибок. Существует три типа ошибок: синтаксические, логические и ошибки времени выполнения (*run-time errors*).

5.1. Как увидеть сообщение об ошибке

Представим, что у нас есть сценарий с ошибкой (листинг 5.1). Сценарий ошибочно пытается обратиться к переменной **Z**, которая объявлена внутри функции.

Листинг 5.1. Сценарий с ошибкой

```
<html>
<head>
    <title>Глобальные и локальные переменные</title>
    <script>
        // Глобальные переменные
        var A = 10;
        var B = 20;

        function F1() {
            // Локальные переменные
            var Z = 10;
            var B = 5;
```

```
document.write("<br>A = " + A);
document.write("<br>B = " + B);
document.write("<br>Z = " + Z);

}

</script>
</head>
<body>
<script>

F1();

document.write("<HR>");
document.write("A = " + A);
document.write("<br>B = " + B);
document.write("<br>Z = " + Z);

</script>
</body>
</html>
```

Современные браузеры скрывают любые сообщения об ошибках JavaScript. В некоторых случаях вы получите просто пустую страницу (без вывода), в некоторых - все, что было выведено до первой ошибки. Так произошло и в нашем случае. Браузеры выводят следующее:

```
A = 10
B = 5
X = 10
-----
A = 10
B = 20
```

Заметьте, последняя строчка ($X = \text{значение}$) выведена не будет вообще. Почему? А потому что произошла ошибка. В нашем простом

сценарии все понятно и без всяких сообщений. Но когда сценарий будет большой и сложный, хотелось бы видеть, какая ошибка произошла и где именно. Для отладки JavaScript-сценариев больше всего подходят браузеры Chrome и Firefox, поскольку у них есть все необходимые для этого средства.

В браузере Chrome/Firefox для вызова средств разработчика нажмите комбинацию клавиш **Ctrl + Shift + I** или **F12** – как кому больше нравится.

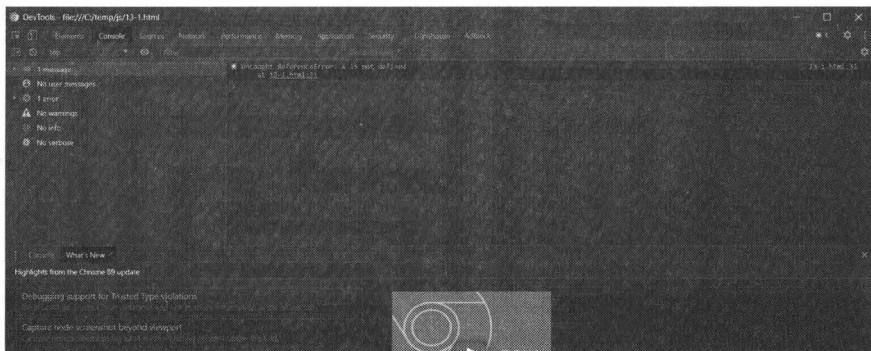


Рис. 5.1. Консоль в Chrome

Браузер Chrome сообщает, что произошла ошибка *Uncaught ReferenceError: X is not defined* (идентификатор *X* не определен) в строке 31 сценария 13-1.html. Браузер Firefox сообщает то же самое: *ReferenceError: X is not defined* (также выводится имя сценария и номер строки). Кроме того, Firefox отображает подробное описание ошибки.

Теперь, когда вы знаете, как увидеть содержание ошибки и ее описание, можно приступить к обзору ошибок, с которым вы можете столкнуться при написании программ.

5.2. Синтаксические ошибки

Синтаксические ошибки - ошибки, касающиеся синтаксиса языка. Вы что-то написали не так, например, забыли закрыть фигурную скобку, допустили ошибку в имени переменной или функции, пробуете использовать неопределенную до этого переменную, забыли где-то поставить точку с запятой и т.д. Обычно интерпретатор уведомляет о таких ошибках, а как просмотреть это уведомление, мы уже знаем. Синтаксические ошибки - самый просто тип ошибки. Интерпретатор не только укажет место, где произошла ошибка, но и подскажет, как ее исправить. Приведенный пример в листинге 5.1 ошибки является именно синтаксической ошибкой: мы попытались обратиться к неопределенной переменной. Переменная то была определена в функции **X**, что находится за пределами видимости для нашего сценария.

Наиболее часто встречающиеся синтаксические ошибки:

- Опечатка в имени переменной или функции.
- Неправильный регистр символов, например, вы обращаетесь к переменной **X**, а объявлена переменная **x** (или наоборот).
- Использование символов национальных алфавитов в названиях функций и переменных, например, символ **X** выглядит одинаково, как в кириллице, так и латинице.
- Отсутствует одна из скобок скобка (круглая, фигурная, квадратная).
- В цикле **for** параметры указаны через запятую, а не через точку с запятой или наоборотнаоборот, в функции параметры указаны через точку с запятой, а не через запятую.
- Вы забыли напечатать точку с запятой в конце оператора.

5.3. Логические ошибки

Бывает так, что с точки зрения синтаксиса программа - идеальна, ошибок нет, но делает она явно не то, что вы от нее хотите. Это и есть пример логической ошибки. Иногда такие ошибки довольно сложно отследить, поскольку это ошибка в алгоритме работы. Тут уже никакой интерпретатор вам не подскажет - во всяком случае, пока интерпретаторы не научились читать мысли.

Найти логическую ошибку можно только посредством тщательного анализа кода программы. Здесь никто кроме вас самих вам не поможет. Один из частых примеров логической ошибки - это ошибка неучтённой единицы. Например, вы можете забыть, что нумерация массива начинается с 0. Рассмотрим следующий код:

```
var M = [1, 2, 3, 4];
for (var i=1; i<4; i++) document.write(M[i]+ " ");
```

С синтаксической точки зрения ошибок нет. Однако вместо того, чтобы вывести:

1 2 3 4

сценарий выведет:

2 3 4

Еще одним примером часто используемой логической ошибки является использование = вместо == в логическом выражении, например:

```
var Y = 10;
```

```
if (Y=11) document.write("Переменная Y = 11");
else document.write("Переменная Y <> 11");
```

Мы хотели проверить, равна ли переменная **Y** одиннадцати. Очевидно, что она не равна 11, поскольку перед этим мы присвоили ей значение 10. Однако, поскольку вместо `==` у нас `=`, то переменной **Y** будет присвоено значение 11, а поскольку процесс присвоения значения прошел успешно (а как он может пройти неуспешно?), оператор `Y = 11` вернет *true*, следовательно, сценарий выведет фразу **Переменная Y = 11**. Кроме того, в результате выполнения этого кода переменной **Y** будет присвоено значение 11.

5.4. Run-time errors или ошибки времени выполнения

Бывает так, что код не содержит ни синтаксических, ни логических ошибок, но не работает так, как запланировано. Ошибки времени выполнения проявляются только во время выполнения сценария. Причина Run-time ошибок - события, которые не предусмотрел программист.

Небольшой пример:

```
if (x>10) window.alert("x > 10");
else windows.alert("x < 10");
```

Если `x > 10`, вы никогда не столкнетесь с ошибкой в этом фрагменте кода, поскольку строка, содержащая ошибку, просто никогда не будет обработана интерпретатором. Если же `x` станет меньше 10, вы столкнетесь с ошибкой времени выполнения: объекта **windows** не существует (есть объект **window**, а программист здесь сделал опечатку, ошибочно добавив `s`). Когда интерпретатор столкнется с ошибкой, выполнение скрипта будет прервано.

5.5. Конструкция try

Конструкция try/catch/finally используется для обработки ошибок, возникающих в процессе выполнения скрипта:

```
try {  
    <Выражения с потенциальными ошибками>  
}  
catch ([<Ссылка на объект Error>]) {  
    <Выражения, которые будут выполнены в случае ошибки>  
}  
finally {  
    <Выражения, которые будут всегда выполнены>  
}
```

В блок **try** помещаются выражения, выполнение которых может порождать ошибку времени выполнения. Ошибки этих выражений мы будем перехватывать. Не нужно стараться поместить в блок **try** весь сценарий. Обычно в него помещаются критические выражения, где есть потенциально высокий риск возникновения ошибок времени выполнения. Если внутри этого блока возникнет исключение, то управление будет передано в блок **catch**. В качестве параметра в блоке **catch** можно указать переменную, через которую будет доступен объект **Error**, содержащий описание ошибки. Если в блоке **try** ошибки не возникло, то блок **catch** не выполняется. Если указан блок **finally**, то выражения внутри этого блока будут выполнены независимо от того, возникла ошибка или нет.

Хотя бы один из блоков **catch** и **finally** должен существовать (допускается использование обоих блоков). Конструкцию try/catch/finally удобно использовать для выполнения операторов, которые должны быть выполнены в любом случае, даже если код в **try** содержит ошибки. Ведь если в коде будет встречена ошибка, выполнение программы будет прервано. А так у нас есть волшебный блок **finally**,

который будет выполнен в любом случае, даже если в блоке **try** есть ошибки, пример:

```
try {
    window.alert('Try');

    t = 10;          // ошибка, переменная не определена
}
finally {
    window.alert('Finally');
}
```

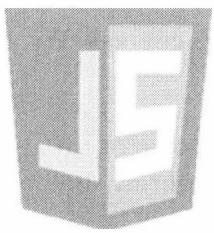
Выполните этот код, и вы увидите оба сообщения: как **Try**, так и **Finally**.

Иногда нужно, наоборот, прервать программу и указать, что произошла ошибка. Для этого используется оператор **throw**. Вот как его нужно использовать:

```
if (d == 0)
    throw new Error("Division by zero");
```

5.6. Метод `console.log()`

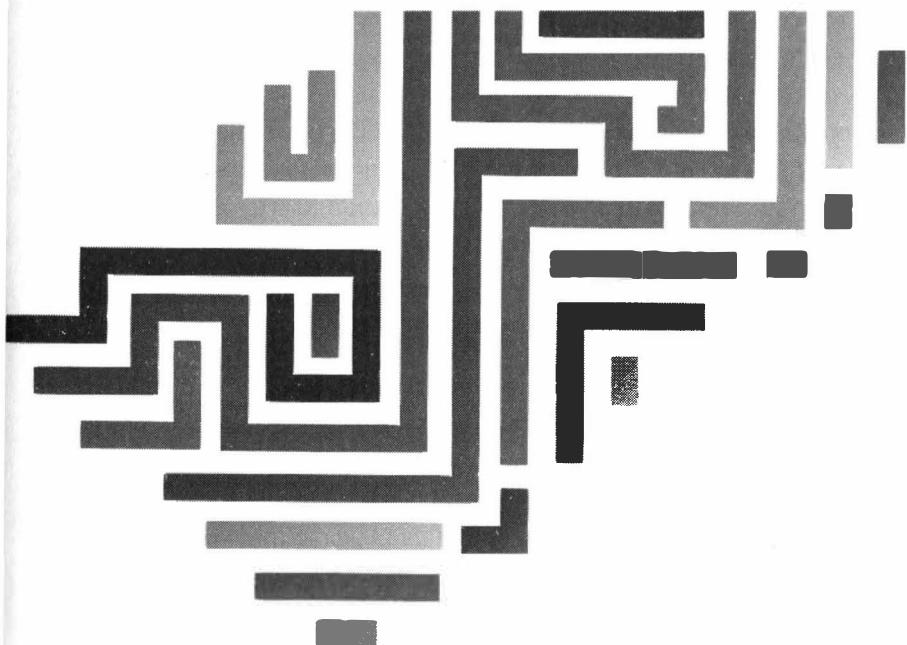
При отладке сценариев старайтесь не использовать метод `window.alert()`, выполнение которого приводит к появлению окна с какой-то информацией. В учебных сценариях его использование допускается, но когда вы вносите изменения в уже существующий сайт, лучше использовать метод `console.log()`, если вам нужно просмотреть какое-то значение. Данное значение будет выведено на консоль браузера (вы его увидите в средствах разработчика на вкладке **Console**) – так и пользователи не увидят то, что не должны, и вы сможете просмотреть, какое значение было присвоено той или иной переменной.



JavaScript

ГЛАВА 6.

Основы объектно-ориентированного программирования на JavaScript



6.1. Основные концепции

6.1.1. Введение в ООП

Объектно-ориентированное программирование (ООП) — это особый подход к написанию программ. Чтобы понять, что такое ООП и зачем оно нужно, необходимо вспомнить некоторые факты из истории развития вычислительной техники. Первые программы вносились в компьютер с помощью переключателей на передней панели компьютера - в то время компьютеры занимали целые комнаты. Такой способ "написания" программы, сами понимаете, был не очень эффективным - ведь большая часть времени (несколько часов, иногда - целый рабочий день) занимало подключение кабелей и установка переключателей. А сами расчеты занимали считанные минуты. Вы только представьте, что делать, если один из программистов (такие компьютеры программировались, как правило, группами программистов) неправильно подключил кабель или установил переключатель? Да, приходилось все перепроверять - по сути, все начинать заново.

Позже появились перфокарты. Программа, то есть последовательность действий, которые должен был выполнить компьютер, наносилась на перфокарту. Пользователь вычислительной машины (так правильно было называть компьютеры в то время) писали программу, оператор "записывал" программу на перфокарту, которая передавалась оператору вычислительного отдела. Через определенное время оператор возвращал пользователю результат работы программы - рулон бумаги с результатами вычислений. Мониторов тогда не

было, а все, что выводил компьютер, печаталось на бумаге. Понятно, если в расчетах была допущена ошибка (со стороны пользователя, компьютеры ведь не ошибаются - они делают с точностью то, что заложено программой), то вся цепочка действий (программист, оператор перфокарты, оператор вычислительной машины, проверка результатов) повторялась заново.

Следующий этап в программировании - это появление языка Ассемблера. Этот язык программирования позволял писать довольно длинные для того времени программы. Но Ассемблер - это язык программирования низкого уровня, все операции проводятся на уровне "железа". Если вы не знаете, то сейчас я вам поясню.

Чтобы в PHP выполнить простейшее действие, например сложение, достаточно записать '\$A = 2 + 2;'. На языке Ассемблера вам для выполнения этого же действия нужно было выполнить как минимум три действия - загрузить в один из регистров первое число (команда MOV), загрузить в другой регистр второе число (опять команда MOV), выполнить сложение регистров командой ADD. Результат сложения будет помещен в третий регистр. Названия регистров я специально не указывал, поскольку они зависят от архитектуры процессора, а это еще один недостаток Ассемблера. Если вам нужно перенести программу на компьютер с другой архитектурой, вам нужно переписать программу с учетом особенностей целевой архитектуры.

Требования к программным продуктам и к срокам их разработки росли (чем быстрее будет написана программа, тем лучше), поэтому появились языки программирования высокого уровня. Язык высокого уровня позволяет писать программы, не задумываясь об архитектуре вашего процессора. Нет, это не означает, что на любом языке высокого уровня можно написать программу, которая в итоге станет работать на процессоре с любой архитектурой. Просто при написании программы знать архитектуру процессора совсем не обязательно. Вы пишете просто $A = B + C$ и не задумываетесь, в каком из регистров

(или в какой ячейке оперативной памяти) сейчас хранятся значения, присвоенные переменным **B** и **C**. Вы также не задумываетесь, куда будет помещено значение переменной **A**. Вы просто знаете, что к нему можно обратиться по имени **A**. Первым языком высокого уровня стал FORTRAN (FORmula TRANslator).

Следующий шаг - это появление структурного программирования. Дело в том, что программы на языке высокого уровня очень быстро стали расти в размерах, что сделало их нечитабельными из-за отсутствия какой-нибудь четкой структуры самой программы. Структурное программирование подразумевает наличие структуры программы и программных блоков, а также отказ от инструкций безусловного перехода (**GOTO**, **JMP**).

После выделения структуры программы появилась необходимость в создании подпрограмм, которые существенно сокращали код программы. Намного проще один раз написать код вычисления какой-то формулы и оформить его в виде процедуры (функции) - затем для вычисления 10 результатов по этой формуле нужно будет 10 раз вызвать процедуру, а не повторять 10 раз один и тот же код. Новый класс программирования стал называться процедурным.

Со временем процедурное программирование постигла та же участь, что и структурное программирование - программы стали настолько большими, что их было неудобно читать. Нужен был новый подход к программированию. Таким стало объектно-ориентированное программирование (далее ООП).

ООП базируется на трех основных принципах - инкапсуляция, полиморфизм, наследование. Разберемся, что есть что.

С помощью инкапсуляции вы можете объединить воедино данные и обрабатывающий их код. Инкапсуляция защищает и код, и данные от вмешательства извне. Базовым понятием в ООП является класс.

Грубо говоря, класс - это своеобразный тип переменной. Экземпляр класса (переменная типа класс) называется объектом. В свою очередь, объект - это совокупность данных (свойств) и функций (методов) для их обработки. Данные и методы обработки называются членами класса.

Получается, что объект - это результат инкапсуляции, поскольку он включает в себя и данные, и код их обработки. Чуть дальше вы поймете, как это работает, пока представьте, что объект - это эдакий рюкзак, собранный по принципу "все свое ношу с собой".

Члены класса могут быть открытыми или закрытыми. Открытые члены класса доступны для других частей программы, которые не являются частью объекта. Закрытые члены доступны только методам самого объекта.

Теперь поговорим о полиморфизме. Если вы программируете на языке С (на обычном С, не C++), то наверняка знакомы с функциями `abs()`, `fabs()`, `labs()`. Все они вычисляют абсолютное значение числа, но каждая из функций используется для своего типа данных. Если бы С поддерживал полиморфизм, то можно было бы создать одну функцию `abs()`, но объявить ее трижды - для каждого типа данных, а компилятор бы уже сам выбирал нужный вариант функции, в зависимости от переданного ей типа данных. Данная практика называется перезагрузкой функций. Перезагрузка функций существенно облегчает труд программиста - вам нужно помнить в несколько раз меньше названий функций для написания программы.

Полиморфизм позволяет нам манипулировать с объектами путем создания стандартного интерфейса для схожих действий.

Осталось поговорить о наследовании. Посредством наследования один объект может приобретать свойства другого объекта. При этом наследование не является копированием объекта. При копировании

создается точная копия объекта, а при наследовании эта копия дополняется уникальными свойствами (новыми членами). Наследование можно сравнить с рождением ребенка, когда новый человек наследует "свойства" своих родителей, но в то же время не является точной копией одного из родителей.

Все выше сказанное было истинно для любого полноценного объектно-ориентированного языка программирования. В JS поддержка ООП довольно ограничена. Классы в JS появились в стандарте ECMAScript 2015, но они представляют собой всего лишь "синтаксический сахар", то есть надстройку над существующим в JS механизмом прототипного наследования. Синтаксис классов не предлагает новую объектно-ориентированную модель, а просто предоставляет более простой способ создания объектов и организации процесса наследования.

Классы в JS являются специальными функциями, поэтому вы можете объявлять их точно так же, как объявляете функции – посредством объявления класса (*class declaration*) и выражения класса (*class expression*).

6.1.2. Объявление класса

Рассмотрим первый способ определения класса – объявление класса. Для этого используется ключевое слово **class** и указывается имя класса, например:

```
class Human {  
    constructor (year, hname) {  
        this.year = year;  
        this.hname = hname;  
    }  
}
```

Разница между объявлением функции (*function declaration*) и объявлением класса (*class declaration*) заключается в том, что объявление функции совершает подъем (*hoisted*), в то время как объявление класса — нет. Поэтому сначала нужно объявить ваш класс и только затем работать с ним, а следующий код сгенерирует исключение типа *ReferenceError*.

```
let Den = new Human(); // Ошибка: ReferenceError
class Human {}
```

6.1.3. Выражение класса

С помощью выражения класса разработчик может создавать именованные и безымянные выражения. В первом случае имя выражения класса находится в локальной области видимости класса и может быть получено через свойства самого класса, а не его экземпляра.

Пример:

```
// безымянный
var Human = class {
    constructor(year, hname) {
        this.year = year;
        this.hname = hname;
    }
};
console.log(Human.name);
// отобразится: "Human"

// именованный
var Den = class Human2 {
    constructor(year, hname) {
        this.year = year;
        this.hname = hname;
    }
};
console.log(Den.name);
// отобразится: "Human2"
```

6.2. Тело класса

В предыдущем разделе было показано, как объявить класс. С одной стороны класс – это просто еще один способ работы с объектом и понятие классов ввели в JS только лишь для наличия самого ключевого слова **class** в языке программирования, чтобы он не выглядел бледно на фоне всех современных языков. С другой стороны, есть официальная документация, стандарт ECMAScript 2015, которому должны следовать все разработчики, в том числе и авторы этой книги.

Тело класса – это часть кода, заключенная в фигурные скобки, которые следует после объявления класса. В теле класса можно объявить члены класса, например, методы и конструктор.

6.2.1. Конструктор

Концепция объектно-ориентированного программирования подразумевает наличие метода-конструктора. Метод **constructor** используется, как правило, для создания и инициализации объектов, которые созданы с помощью класса. Разумеется, внутри одного класса может быть только один метод-конструктор.

Внимание! Если в классе описано более одного конструктора, вы получите исключение *SyntaxError*.

Для вызова конструктора родительского класса в методе **constructor** используется ключевое слово **super**.

6.2.2. Статические методы и свойства. Ключевое слово **static**

Статические методы и свойства вызываются без инстанцирования их класса, и не могут быть вызваны у экземпляров (*instance*) класса.

Объявить статический член класса возможно с помощью ключевого слова **static**.

Статические методы используются для создания служебных функций для приложения, а статические свойства полезны для кеширования в рамках класса, фиксированной конфигурации или любых других целей, не связанных с реплицированием данных между экземплярами.

Рассмотрим пример:

```
class Human {  
    constructor(year, hname) {  
        this.year = year;  
        this.hname = hname;  
    }  
    static displayName = "Человек";  
}  
  
const p1 = new Human(1983, "Den");  
p1.displayName; //undefined  
console.log(Human.displayName); // Человек
```

6.2.3. Использование **this** в прототипных и статических методах

Если статический или прототипный метод вызывается без привязки к **this** объекта (или когда **this** является типом boolean, string, number, undefined, null), то оператор **this** будет иметь значение *undefined* внутри вызываемой функции.

Пример:

```
class Human {  
    scream() {
```

```

        return this;
    }
    static listen() {
        return this;
    }
}

let obj = new Human();
obj.scream(); // объект Human
let scream = obj.scream;
scream();      // ошибка undefined

Human.listen() // класс Human
let listen = Human.listen;
listen();      // ошибка undefined

```

6.2.4. Свойства экземпляра

Свойства экземпляра должны быть определены в конструкторе:

```

class Box {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
}

```

6.2.5. Публичные и приватные поля

В JS появились публичные и приватные поля, но это экспериментальная функция, предложенная комитетом TC39. Пока они не поддерживаются текущими версиями браузеров и лучше отказаться от их использования. Пример объявления публичных полей:

```

class Box {
    height = 0;
    width;
}

```

```
constructor(height, width) {  
    this.height = height;  
    this.width = width;  
}  
}
```

Приватные поля объявляются с помощью решетки:

```
class Box {  
    #height = 0;  
    #width;  
    constructor(height, width) {  
        this.#height = height;  
        this.#width = width;  
    }  
}
```

6.2.6. Наследование. Оператор extends

Ключевое слово **extends** используется для создания класса, дочернего относительно другого класса. Попросту говоря – для наследования.

Пример:

```
class Human {  
    constructor(name) {  
        this.name = name;  
    }  
  
    speak() {  
        console.log(` ${this.name} говорит.`);  
    }  
}  
  
class Vasya extends Human {  
    constructor(name) {  
        super(name); // вызывает конструктор super класса и  
                    // передает параметр name
```

```
}

speak() {
    console.log(` ${this.name} говорит.`);
}
}

let v = new Vasya('Василий');
v.speak(); // Василий говорит
```

6.3. Создание пользовательских объектов

Мы уже знакомы немного с классами и объектами. Создать новый объект можно с помощью функции-конструктора **Object**, например:

```
var Human = new Object();
// свойства объекта
Human.firstname = "John";
Human.lastname = "Doe";
// Формируем метод объекта
Human.getFullName = function() {
    var fname = this.firstname + this.lastname;
    return fname;
}

// просто выводим значения:
window.alert(Human.firstname);
window.alert(Human.lastname);
window.alert(Human.getFullName);
```

После создания объекта в переменной **Human** сохраняется на него ссылка. В качестве значения свойства объекта можно использовать любой тип данных (строку, массив, число и др.). Если в качестве значения указана ссылка на функцию, то такое свойство становится

методом объекта, внутри которого доступен указатель на текущий объект (`this`).

Если вы привыкли к другим языкам программирование и такое создание объекта вам не привычно, вы можете использовать фигурные скобки для определения свойств и методов объекта:

```
var Human = {  
    firstname: "John";  
    lastname: "Doe";  
    getFullName: function() {  
        var fname = this.firstname + this.lastname;  
        return fname;  
    }  
}
```

Значение свойств/методов указывается через двоеточие. Создать пустой объект можно с помощью фигурных скобок:

```
var empty_obj = {};
```

Представим, что нужно создать два одинаковых объекта, которые должны использоваться раздельно. Новички допускают ошибку и используют следующее объявление:

```
var ob1 = ob2 = {};
```

Они допускают очень грубую ошибку: данный оператор создает только один объект и две ссылки на него, которые будут храниться в переменных `ob1` и `ob2`. Все изменения `ob1` будут отражены в `ob2` и наоборот. Пример:

```
ob1.firstname = "Марк";  
document.write(ob2.firstname); // будет выведено Марк
```

Выход только один – использовать два разных оператора присваивания – только так. Пример:

```
var o1 = {};
var o2 = {};
```

Если после ключевого слова **new** указать функцию, то она станет конструктором объекта. Такой функции можно передать начальные данные для инициализации объекта (можно ничего не передавать, все зависит от реализации). Функции-конструкторы удобно использовать, если нужно инициализировать несколько подобных объектов. Рассмотрим функцию конструктор:

```
function Human(firstname, lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
    this.getFullName = function() {
        var fname = this.firstname + this.lastname;
        return fname;
    }
}

var John = new Human("John", "Doe");
var Ivan = new Human("Ivan", "Ivanov");
```

В JS есть удобный цикл **for..in**, позволяющий пройтись по всем свойствам объекта. Это позволяет вывести свойства объекта, которые ранее были неизвестны. А так как у объектов нет закрытых свойств, то будут выведены абсолютно все свойства объекта. Пример:

```
for (var P in John) {
    document.write(P + " = " + John[P]);
}
```

Обратите внимание: мы обращаемся к объекту, как к массиву (в PHP такие массивы называются ассоциативными, где в качестве индекса может использоваться не только число, но и строка).

Оператор **in** позволяет проверить существование свойства в объекте, например:

```
if ("firstname" in John) window.alert(John.firstname);
```

Проверить наличие метода можно, указав его имя без скобок:

```
if (John.getFullName) window.alert('getFullName exists');
```

Так нельзя проверять наличие свойства, поскольку значение `0` будет интерпретироваться как `false`, в итоге такой проверки вы получите, что свойство не существует, но на самом деле оно существует, но просто равно `0`.

Оператор **instanceof** позволяет проверить принадлежность экземпляра классу, например:

```
if ((typeof John == "object") && (John instanceof Human))
    window.alert('John - экземпляр Human');
```

Удалить свойство можно так:

```
delete Ivan.lastname;
```

6.4. Прототипы

Ранее мы определяли метод `getFullName()` внутри конструктора:

```
function Human(firstname, lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
    this.getFullName = function() {
        var fname = this.firstname + this.lastname;
        return fname;
    }
}
```

Такое решение не всегда эффективно. Например, нужно создать массив, состоящий из 1000 объектов. При этом свойства будут разные для всех объектов, а метод `getFullName()` будет одинаковый для всех.

Прототипы позволяют определить метод вне конструктора. При создании объекта он унаследует все свойства, определенные в прототипе. Поэтому метод `getFullName` будет определен всего один раз, но будет унаследован всеми экземплярами класса.

Рассмотрим пример:

```
function Human(firstname, lastname) {  
    this.firstname = firstname;  
    this.lastname = lastname;  
}  
  
Human.prototype.getFullName = function() {  
    var fname = this.firstname + this.lastname;  
    return fname;  
}  
  
var John = new Human("John", "Doe");  
document.write(John.getFullName()); // John Doe  
  
var Ivan = new Human("Ivan", "Ivanov");  
document.write(Ivan.getFullName()); // Ivan  
Ivanov
```

6.5. Пространства имен

Представим, что вы написали функцию `DecodeString()`. Затем вы подключили какую-то библиотеку, в которой есть функция с таким же названием. Произойдет конфликт имен. В результате будет использована та функция, которая была определена последней. Ясно,

что параметры этих двух функций могут отличаться, и если вы попытаетесь вызвать свою функцию, указав 3 параметра, а последней была определена функция `DecodeString()` с двумя параметрами, выполнение сценария будет остановлено с ошибкой.

Чтобы этого не произошло, используются пространства имен. Каждая функция будет определена в своем пространстве имен, а вам нужно будет указывать, из какого пространства имен вызывается та или иная функция.

В JavaScript в качестве пространств имен используются объекты. Созданный экземпляр объекта будет помещен в глобальную область видимости. Все остальные идентификаторы будут доступны через свойства объекта, например:

```
var MyLibrary = {};
MyLibrary.DecodeString = function() {
    document.write('Test');
}
MyLibrary.DecodeString();
```

В данном примере функция `DecodeString()` определена внутри пространства имен `MyLibrary`. Понятно, что конфликт имен сводится к минимуму (конечно, если вы не вздумали назвать свой объект так, как называется сторонняя библиотека).

Если вы занимаетесь JS-разработкой профессионально и пишите скрипты для разных сайтов, рекомендуется создавать пространства имен, совпадающие с именем сайта. Так вы полностью исключите возможность конфликта именно, а чтобы было удобнее обращаться к свойствам и методам, используйте доступ по ссылке. Например:

```
var nitcenter = {};           // создаем пространство имен
var $ = nitcenter;           // создаем ссылку на
пространство имен
```

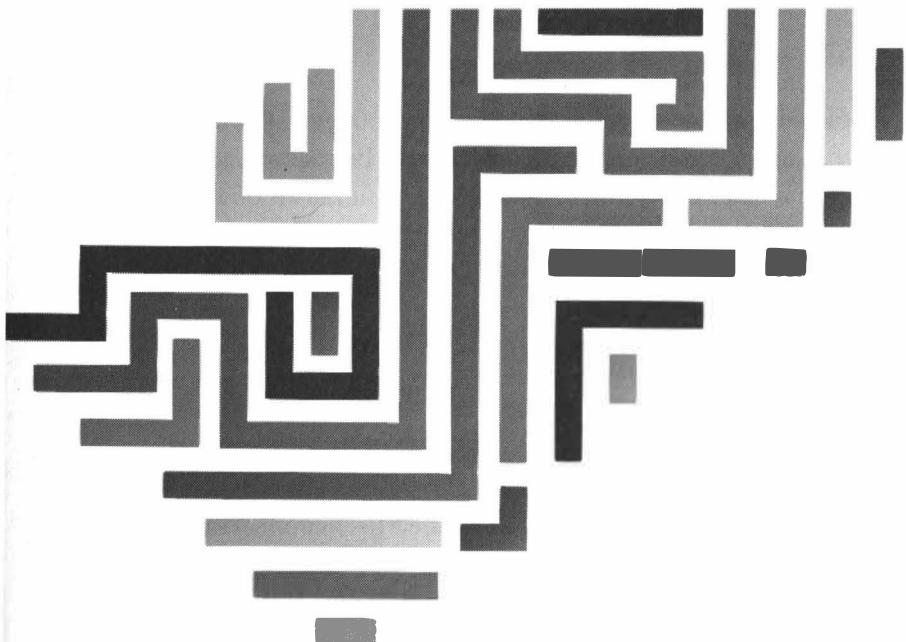
Далее используем ссылку `$` (чтобы не переписывать весь код, когда вам нужно реализовать подобные функции на разных сайтах), например:

```
$ DecodeString();
```

В этой главе мы разобрались, как создавать собственные классы и объекты в JS. В следующей главе мы рассмотрим встроенные классы JS.

ГЛАВА 7.

Встроенные объекты JavaScript



В JavaScript имеется богатый набор встроенных (стандартных) объектов, содержащих множество полезных методов, которые вы можете использовать при написании своих программ. Если вы пропустили предыдущую главу, настоятельно рекомендую с ней ознакомиться, поскольку она нужна для понимания изложенного в этой главе материала.

7.1. Объект Global

Чтобы использовать свойства и методы объекта **Global** вам не нужно создавать экземпляр этого объекта. Все методы этого объекта доступны как встроенные функции. Свойствами объекта являются **NaN** (Not a Number, не число) и **Infinity** (плюс бесконечность), которые вы можете использовать в качестве значений. Например:

```
var x = NaN;  
var y = Infinity;
```

Методы объекта Global:

- **parseInt(<строка>, <основание>)** – используется для преобразования строки в целое число, система счисления числа задается вторым параметром. Если он отсутствует, используется десятичная система. Если строка не может быть преобразована в число, возвращается значение **Nan**.

- **parseFloat(<строка>)** – используется для преобразования строки в число с плавающей точкой.
- **eval(<строка>)** – производит вычисление выражения, заданного в строке, как если бы оно было обычным выражением JavaScript;
- **isNaN(<выражение>)** – позволяет проверить, является ли выражение правильным числом. Если число не является правильным числом (NaN), возвращается *true*, *false* – в противном случае
- **isFinite(<выражение>)** – позволяет проверить, является ли выражение конечным числом (возвращает *true* или *false*). Возвращает *true* для обычных чисел и *false* для значения **Infinity**.
- **escape(<строка>)** – используется для Escape-кодирования строки (кодирует строку шестнадцатеричными символами).
- **unescape(<строка>)** – выполняет обратное преобразование строки.
- **encodeURI(<строка>)** – используется для URI-кодирования строки (подобное мероприятие полезно при формировании URI, содержащего символы национальных алфавитов).
- **decodeURI(<строка>)** – выполняет обратное кодирование строки.

Рассмотрим несколько примеров:

```
var str = "100";
var x = 50 + parseInt(str);                                // 150
var strf = "100.52";
var y = parseFloat(strf);                                 //100.52
var z = eval("2 + 2");                                    // 4
```

7.2. Объект Number

Объект **Number** используется для работы с числами. Экземпляр объекта можно создать так:

```
var <объект> = new Number(начальное значение);
```

Например:

```
var z = Number(1000);
```

Свойства объекта **Number**, которые можно использовать без создания экземпляра объекта:

- **MAX_VALUE** – число, максимальное допустимое в JavaScript
- **MIN_VALUE** – минимальное число для JavaScript
- **NaN** – значение NaN
- **NEGATIVE_INFINITY** – значение "минус бесконечность"
- **POSITIVE_INFINITY** – значение "плюс бесконечность"

У объекта **Number** всего два метода:

- **valueOf()** – возвращает числовое значение экземпляра объекта
- **toString()** – позволяет преобразовать числовое значение объекта в строку

Примеры:

```
var myUniqueVariableName1 = Number.MAX_VALUE;  
var myUniqueVariableName2 = new Number(1000);  
var str = myUniqueVariableName2.toString();           //  
"1000"
```

7.3. Объект String

Объект **String** используется для обработки строк. Экземпляр объекта можно создать так:

```
var <объект> = new String (<строка>);
```

Например:

```
var so = new String ("Привет всем");
```

Но мы уже знаем, что строку можно создать гораздо проще, например:

```
var s = "Привет всем";
```

Разница в том, что тип переменной **s** будет **string**, а переменной **so** - **object**. Однако к обычным строкам можно применять методы объекта **String**, например:

```
var str = "Hello, world!".toUpperCase();
```

Если используется метод **toUpperCase()**, строка, имеющая тип данных **string**, будет автоматически преобразована в объект объекта **String**. Объект **String** является оберткой над типом данных **string**.

У объекта **String** есть всего одно свойство - **length**, содержащее длину строки:

```
var so = new String ("Hello");
document.write(so.length); // 5
```

А вот методов у объекта **String** значительно больше, поэтому они приведены в таблице 7.1 для большего удобства чтения.

Таблица 7.1. Методы объекта String

Метод	Назначение
<code>toString()</code>	Используется для преобразования объекта String в строку
<code>valueOf()</code>	Возвращает значение, хранящееся в объекте строки
<code>charAt(<номер символа>)</code>	Возвращает символ строки с указанным номером. Нумерация символов начинается с 0. Пример: <code>var s = "Hello"; var x = s.charAt(0); // "H"</code>
<code>charCodeAt</code>	Возвращает код символа строки с указанным номером. Нумерация начинается с 0. Пример: <code>var s = "Hello"; var x = s.charCodeAt(0); // 72</code>
<code>fromCharCode (<код1>, ..., <кодN>)</code>	Позволяет создать строку из указанных кодов
<code>toLowerCase()</code>	Используется для преобразования символов строки в нижний регистр. Пример: <code>var myCarName = "Tatra"; s = myCarName.toLowerCase(); // tatra</code>
<code>toUpperCase()</code>	Преобразует символы строки в символы верхнего регистра

<pre>substr(<начало фрагмента>, [длина])</pre>	<p>Позволяет извлечь фрагмент строки заданной длины. Если второй параметр пропущен, будут возвращены все символы до конца строки.</p> <pre>var S = "Hello, dummy"; console.log(S.substr(0, 5)); // "Hello" console.log(S.substr(7)); // "dummy"</pre>
<pre>substring(<начало фрагмента>, <конец фрагмен- та>)</pre>	<p>Позволяет извлечь фрагмент строки, который задан номерами начального и конечного символов. Последний символ в подстроку не включается, например:</p> <pre>var S = "Hello, dummy"; console.log(S.substring(7, 12)); // "dummy"</pre>
<pre>indexOf(<подстрока>, [<на- чальная позиция поиска>])</pre>	<p>Возвращает номер позиции первого вхождения подстроки в текущей строке. Если подстрока не найдена, возвращает -1. Пример:</p> <pre>var S = "Hello, dummy"; console.log(S.indexOf("llo")); // 2</pre>
<pre>lastIndexOf(<подстрока>, [<начальная позиция поис- ка>])</pre>	<p>Возвращает номер позиции последнего вхождения подстроки в текущей строке. Если подстрока не найдена, возвращает -1.</p>

<pre>split(<Разделитель>, [<Лимит>])</pre>	<p>Используется для разделения строки на подстроки по символу-разделителю и возвращает массив. Например:</p> <pre>var S = "Hello, dummy"; var M = S.split(","); console.log(M[0]); // "Hello" console.log(M[1]); // " dummy"</pre>
<pre>search(<регулярное выражение>)</pre>	<p>Позволяет определить номер позиции первого вхождения подстроки, совпадающей с регулярным выражением</p>
<pre>match(<регулярное выражение>)</pre>	<p>Возвращает массив с результатами поиска, которые совпадают с регулярным выражением</p>
<pre>replace(<регулярное выражение>, <текст для замены>)</pre>	<p>Используется для поиска и замены в исходной строке с использованием регулярного выражения.</p>

7.4. Объект Array

7.4.1. Свойства и методы объекта

Ранее мы говорили о создании массивов. Сейчас мы рассмотрим объект **Array**, который можно использовать для обработки массивов, а также для их создания. Создать экземпляр этого объекта можно так:

```
var <объект> = new Array (<размер массива>);
var <объект> = new Array (<элементы массива через запятую>);
```

Пример:

```
var M = new Array(1, 2, 3);
```

Свойство **length** содержит количество элементов массива. Нумерация элементов массива начинается с 0. Пример использования свойства **length**:

```
document.write(M.length);

for (var i=0, c=M.length; i<c; i++) {
    document.write(M[i] + "<br>");
}
```

Методов у этого объекта не меньше, чем у объекта **String**. Методы приведены в табл. 7.2.

Таблица 7.2. Методы объекта Array

Метод	Назначение
push (<элементы>)	Позволяет добавить элементы в конец массива, возвращает новую длину массива: M.push(4, 5);
unshift (<элементы>)	Позволяет добавить элементы в начало массива
concat (<элементы>)	Возвращает массив, полученный в результате объединения текущего массива и списка элементов. В текущий массив элементы из списка не добавляются. Например: <pre>var M1 = new Array(1, 2, 3); var M2 = []; M2 = M.concat(4, 5); // M1 = [1, 2, 3] // M2 = [1, 2, 3, 4, 5]</pre>

join(<разделитель>)	Объединяет элементы массива в строку, разделяя их заданным разделителем. Пример: <pre>var Massiv = new Array(5, 6, 7); console.log(Massiv.join(" ")); // 5 6 7</pre>
shift()	Выполняет сдвиг, то есть возвращает первый элемент массива и удаляет его из массива
pop()	"Выталкивает" последний элемент массива и удаляет его из массива
sort(<функция сортировки>)	Используется для сортировки массива указанной функцией сортировки. Если она не указана, будет выполнена обычная сортировка (числа - по возрастанию, символы - по алфавиту)
reverse()	Переворачивает массив. Элементы массива будут в обратном порядке
slice(<начало>, [конец])	Возвращает срез массива, начиная от индекса <начало> до индекса <конец>. Если второй индекс не задан - до конца массива. Пример: <pre>var M1 = new Array(1, 2, 3, 4, 5, 6); var M2 = M1.slice(1, 3); // [2, 3]</pre>
toString()	Позволяет преобразовать массив в строку. Элементы указываются через запятую без пробела.

7.4.2. Сортировка массива

Отдельного внимания заслуживает сортировка массива. Функция `sort()` способна выполнить только базовую сортировку массива, которой не всегда достаточно, поэтому программист может создать собственную функцию-сортировки. Такая функция должна принимать две переменные и возвращать:

- 0 если обе переменные равны;
- -1 если вторая переменная больше первой;
- 1 если первая переменная больше второй.

Рассмотрим пример сортировки строк без учета регистра:

```
function my_unique_sort(first, second) {  
    var x = first.toLowerCase(); // к нижнему регистру  
    var y = second.toLowerCase(); // к нижнему регистру  
    if (x > y) return 1;  
    if (x < y) return -1;  
    return 0;  
}  
  
var Mass = [ "One", "two9", "open" ];  
Mass.sort(my_unique_sort);  
console.log(Mass.join(", "));
```

Разработчик может изменить порядок сортировки, поменяв возвращаемые значения на противоположные:

```
function my_unique_sort(first, second) {  
    var x = first.toLowerCase(); // к нижнему регистру  
    var y = second.toLowerCase(); // к нижнему регистру  
    if (x > y) return -1;  
    if (x < y) return 1;  
    return 0;  
}
```

7.4.3. Многомерные массивы

Многомерные массивы можно создать несколькими способами. Например, можно их создавать поэлементно:

```
var M = [];
M[0] = [];
M[1] = [];
M[0][0] = 1;
M[0][1] = 2;
M[0][2] = 3;
M[1][0] = 3;
M[1][1] = 2;
M[1][2] = 3;
```

Также можно использовать перечисление, например:

```
var M = new Array(new Array("1", "2", "3"),
                  new Array("3", "2", "1"));
```

7.4.4. Ассоциативные массивы

Если вы программировали на PHP, то наверняка знакомы с ассоциативными массивами. Ассоциативные массивы позволяют в качестве индексов использовать строки, а не только числа. Пример:

```
var M = new Array();
M ["one"] = 1;
M ["two"] = 2;
```

Методы объекта `Array` не позволяют отобразить элементы ассоциативного массива. Свойство `length` также недоступно, поэтому нет

возможности перебрать элементы ассоциативного массива в цикле **for**. Для этого нужно использовать цикл **for ..in**:

```
for (let N in M) {  
    document.write(N + " = " + M[N] + "<br>");  
}
```

Вывод будет таким:

```
one = 1  
two = 2
```

7.5. Объект Math

Объект **Math** содержит некоторые математические функции и константы. Использование этого объекта не требует создания экземпляра объекта.

В объекте Math содержатся следующие константы:

- E – экспонента, основание натурального логарифма
- LN2 – натуральный логарифм 2
- LN10 – натуральный логарифм 10
- LOG2E – логарифм по основанию 2 от E
- LOG10E – логарифм по основанию 10 от E
- PI – число Пи
- SQRT2 – квадратный корень из 2
- SQRT1_2 – квадратный корень из 0.5

Методы объекта Math представлены в таблице 7.3.

Таблица 7.3. Методы объекта Math

Метод	Назначение
abs()	Возвращает абсолютное значение числа
sin(), cos(), tan(), asin(), acos(), atan()	Стандартные тригонометрические функции
exp()	Экспонента
log()	Натуральный логарифм
pow(<число>, <степень>)	Возведение в степень
sqrt()	Квадратный корень
round()	Значение, округленное до ближайшего целого. Округление может осуществляться, как в большую, так и меньшую сторону
ceil()	Значение, округленное до ближайшего большего целого.
floor()	Округление до ближайшего меньшего целого.
max(<список элементов через запятую>), min(<список элементов через запятую>)	Методы возвращают максимальное/минимальное значение из списка соответственно
random()	Возвращает случайное число от 0 до 1.

Как использовать математические функции на практике? Пусть у нас есть 4 баннера и нужно их выводить случайным образом при обновлении страницы. Вот пример кода:

```
var nu = Math.floor(Math.random()*5.9999);
document.write('');
```

Файлы с баннерами должны называться banner0.jpg...banner3.jpg.

7.6. Объекты Function и Arguments

Объект **Function** позволяет использовать функцию в качестве экземпляра объекта:

```
<имя функции> = new Function(<аргумент 1>, ...,
                                <аргумент N>, <код>);
```

Пример:

```
var Sum = new Function("x", "y", "return x+y");
```

Однако таким способом мало кто пользуется, поскольку указывать код функции в виде строки очень неудобно. Зато можно использовать анонимные функции, например:

```
var Sum = function(x, y) { return x + y; }
```

Вызвать функцию можно, как и раньше:

```
window.alert(Sum(2, 2));
```

В JavaScript разработчик может создавать функции с произвольным числом аргументов. При этом ему доступен массив **arguments**, через который он может получить доступ к аргументам функции, а также

свойство **length** массива **arguments** – содержит количество переданных функции параметров. Напишем функцию SumArgs, которая вычисляет сумму произвольного числа аргументов:

```
function SumArgs() {
    var r = 0;
    for (var i=0; i < arguments.length; i++) r = r + arguments[i];
    return r;
}
console.log(SumArgs(1, 2, 3));
```

7.7. Объект Date

Для работы с датой и временем в JavaScript используется объект **Date**. Есть несколько способов создать экземпляры этого объекта:

```
var <объект> = new Date();
var <объект> = new Date(<количество миллисекунд>);
var <объект> = new Date(<год>, <месяц>, <день>, <часы>,
<мин>, <c>, <мс>);
```

Объект Date содержит много методов, которые вместе с примерами по их использованию, приведены в таблице 7.4.

Таблица 7.4. Методы объекта Date

Метод	Описание
toString()	<p>Преобразует дату в строку и возвращает ее. Пример:</p> <pre>var d = new Date(); // текущая дата console.log(d.toString());</pre> <p>Вывод может отличаться в зависимости от браузера. В Chrome вывод будет таким:</p> <pre>Sat Apr 03 2021 10:48:37 GMT+0300 (Восточная Европа, летнее время)</pre>

toLocaleDateString()	Преобразует дату в строку с использованием интернациональных установок системы. Параметры моей системы таковы, что Chrome отобразил строку: 30.1.2014 15:38:57
valueOf()	Возвращает число секунд, прошедших с 01.01.1970 00:00:00. <code>var d = new Date(); // текущая дата console.log(d.valueOf());</code> Вывод: 1617436169721
getDate()	День месяца (от 1 до 31)
getDay()	Позволяет получить день недели, 0 - воскресение, 1 - понедельник и т.д.
getMonth()	Номер месяца (0 - январь, 11 - декабрь)
getFullYear()	Позволяет получить полный год (например, 2014)
getHours()	Час (от 0 до 23)
getMinutes()	Минуты (от 0 до 59)
getSeconds()	Секунды (от 0 до 59)
getMilliseconds()	Возвращает миллисекунды (от 0 до 999)
getTime()	Возвращает то же значение, что и valueOf()

Напишем небольшую функцию, возвращающую название месяца по переданному ей номеру месяца:

```
function getStrMonth(m) {  
  
    var d = new Date();  
  
    var MonthsArray = [ "Янв", "Фев", "Мар", "Апр", "Май",  
        "Июн", "Июл", "Авг", "Сен", "Окт", "Ноя", "Дек" ];  
  
    var currentMonth = d.getMonth();  
  
    if ((m < 0) || (m > 11)) return  
        MonthsArray[CurrentMonth];  
  
    return MonthsArray[m]; // нумерация массива с 0  
}  
  
var d = new Date();  
console.log(getStrMonth(d.getMonth()));
```

Функция `getStrMonth()` определяет массив `MonthsArray`, который содержит символьные названия месяцев. Функция определяет текущий месяц - на случай, если она будет вызвана с неправильным номером месяца (который меньше 0 или больше 11). В этом случае функция вернет название текущего месяца (а не сообщение об ошибке - так лучше, чем выводить сообщение об ошибке, которое увидит пользователь).

Рассмотрим использование функции:

```
var d = new Date();  
console.log(getStrMonth(d.getMonth()));
```

Этот код можно смело заменить на такой код:

```
console.log(getStrMonth(-1));
```

Как видите, функцию `getStrMonth()` можно использовать, как в паре с методом `getMonth` (что было продемонстрировано выше), так и напрямую – путем передачи ей номера месяца (нумерация месяцев начинается с 0!).

7.8. Объект RegExp

Организация поиска в строке посредством регулярных выражений возможна путем использования объекта `RegExp`. Смотрите на регулярные выражения как на шаблоны для поиска определенных комбинаций мета-символов. Они используются для осуществления сложного поиска в строке, когда нужно найти не просто строку, а строку, соответствующую определенному шаблону, например, номер телефона, e-mail и т.д.

Работа с `RegExp` начинается с создания экземпляра объекта:

```
var <объект> = new RegExp(<регулярное выражение>,
                           <модификатор>);
var <объект> = /<регулярное выражение>/[<модификатор>];
```

Модификатор может принимать следующие значения:

- `i` – поиск без учета регистра;
- `g` – глобальный поиск;
- `m` – многострочный поиск;
- `gi` – глобальный поиск без учета регистра.

Регулярные выражения, поддерживаемые в `JavaScript`, используются и в других языках программирования, например, в `PHP`, что до-

бавит вашему коду больше универсальности. Документация по регулярным выражениям доступна по адресу:

https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions

Чтобы заинтересовать читателя, мы приведем несколько примеров использования регулярных выражений. Начнем с метода `search()`, возвращающего номер позиции первого вхождения подстроки, которая совпадает с регулярным выражением:

```
let reg = new RegExp("abc[de]");  
let Str = "abc, abcd, abce, abcf";  
console.log(Str.search(reg));           // 5
```

В результате будет выведено 5, поскольку первая подстрока, соответствующая регулярному выражению "abc[de]", начинается с позиции 5 (шестой символ, нумерация с 0). Вообще в строке `Str` есть две подстроки, соответствующих заданному регулярному выражению - **abcd** и **abce**, но функция возвращает позицию только первой найденной подстроки.

Когда нужно найти все соответствия, используйте метод `match()`, возвращающий массив с результатами поиска:

```
let reg = new RegExp("abc[de]");  
let Str = "abc, abcd, abce, abcf";  
let M = [];  
  
M = Str.match(reg);  
  
for (var i=0, c=M.length; i<c; i++)  
    console.log(M[i] + " ");
```

Этот пример выведет только **abcd**, чтобы `match()` вывел остальные варианты, нужно использовать модификатор глобального поиска:

```
var p = new RegExp("abc[de]", "g");
```

Вот тогда результат будет таким, как нужно:

```
abcd abce
```

Метод `replace(<регулярное выражение>, <текст для замены>)` позволяет произвести поиск и замену текста с использованием регулярного выражения.

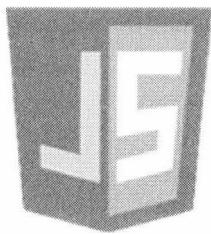
```
let reg = new RegExp("abc[de]", "g");
let Str = "abc, abcd, abce, abcf";
let S = Str.replace(reg, "match");
console.log(S);
```

В результате будет выведена строка:

```
abc, match, match, abcf
```

Все найденные подстроки, соответствующие регулярному выражению, будут заменены на "match".

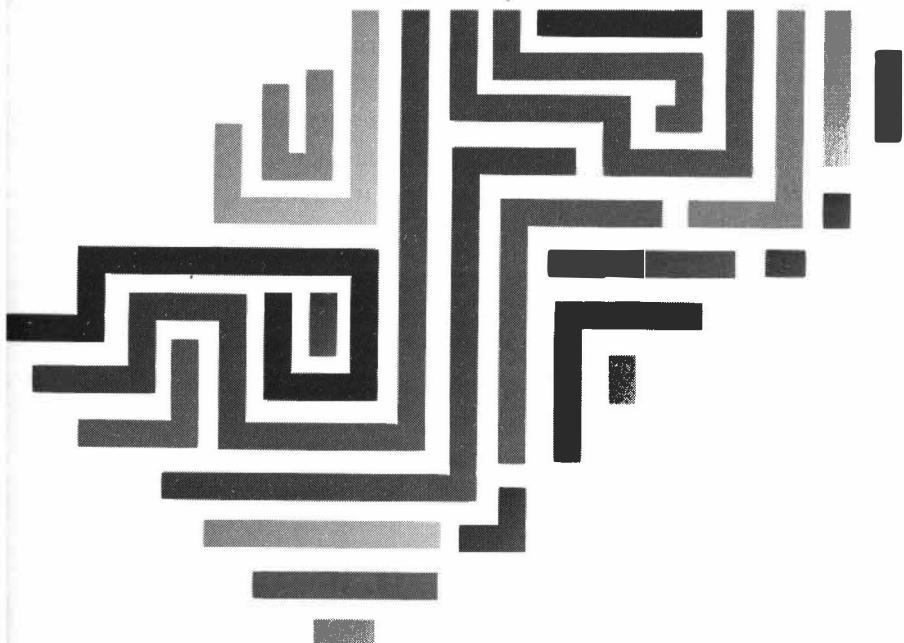
На этом мы заканчиваем рассмотрение встроенных объектов JavaScript и переходим к рассмотрению событий, которые будут рассмотрены в следующей главе.



JavaScript

ГЛАВА 8.

События в JavaScript



8.1. Что такое событие?

События происходят при взаимодействии пользователя с веб-страницей. Посредством событий система извещает сценарий, что пользователь выполнил какое-то действие: переместил мышь, нажал кнопку мыши, нажал какие-то клавиши на клавиатуре, изменил размеры окна и т.д. Например, при нажатии кнопки мыши возникает событие `onmousedown`.

8.2. События мыши

События мыши описаны в таблице 8.1.

Таблица 8.1. События мыши

Событие	Когда происходит
onmousedown	Происходит, когда пользователь нажимает кнопку мыши на элементе веб-страницы или самой странице. Именно нажимает кнопку, а не нажимает и отпускает ее – это событие onclick .
onmouseup	Возникает при отпускании ранее нажатой кнопки мыши.
onclick	Происходит, когда нажимает и отпускает кнопку мыши (делает щелчок) на элементе веб-страницы или самой странице
ondblclick	Возникает, когда пользователь делает двойной клик на элементе страницы
onmouseover	Генерируется при наведении курсора мыши на элемент страницы (картинку, надпись, абзац, кнопку и т.д.)
onmouseout	Когда курсор мыши выходит за пределы элемента страницы
onmousemove	При перемещении мыши (при любом перемещении)
onselect	При выделении элемента
onselectstart	При начале выделения
oncontextmenu	При выводе контекстного меню (когда пользователь нажимает правую кнопку или левую кнопку, если пользователь - левша).

8.3. События клавиатуры

Аналогично, события клавиатуры представлены в табл. 8.2.

Таблица 8.2. События клавиатуры

Событие	Когда происходит
onkeydown	Возникает при нажатии клавиши на клавиатуре
onkeypress	Генерируется при нажатии клавиши на клавиатуре, но при этом возвращает код нажатого символа в Unicode
onkeyup	Когда пользователь отпускает нажатую ранее клавишу
onhelp	Возникает при нажатии клавиши F1

8.4. События документа

Таблица 8.3 содержит основные события документа.

Таблица 8.3. События документа

Событие	Когда происходит
onload	Генерируется сразу после загрузки веб-страницы
onscroll	Возникает при прокручивании содержимого страницы
onresize	Возникает при изменении размеров окна
onunload	Генерируется при выгрузке документа. Наступает после события onbeforeunload

onbeforeunload	Перед выгрузкой документа, перед событием <code>onunload</code>
onbeforeprint	Перед распечаткой документа
onafterprint	После распечатки документа

8.5. События формы

События формы представлены в таблице 8.4.

Таблица 8.4. События формы

Событие	Когда происходит
onsubmit	При отправке формы (при нажатии кнопки Submit)
onreset	При сбросе формы (при нажатии кнопки Reset)
onblur	При потере фокуса элементом формы
onfocus	Когда элемент формы (например, поле ввода или кнопка) получает фокус (становится активным)
onchange	Когда фокус перемещается на другой элемент кнопки (наступает перед <code>onblur</code>)

8.6. Последовательность событий

События возникают в определенной последовательности, например, при нажатии кнопки мыши последовательность будет следующей: `onmousedown`, `onmouseup`, `onclick`.

При двойном клике мыши последовательность будет немного другой: `onmousedown`, `onmouseup`, `onclick`, `ondblclick`.

Последовательность событий нужно учитывать при установке обработчиков событий - в зависимости от желаемого результата.

Рассмотрим сценарий (листинг 8.1), демонстрирующий последовательность событий. В нем мы устанавливаем обработчики событий (подробно о них мы поговорим в р. 8.9), чтобы проследить их последовательность.

Листинг 8.1. Пример обработки событий мыши

```
<html>
<head>
<title>События</title>
</head>
<body onload="window.alert('OnLoad. Нажмите кнопку
мыши');"
      onmousedown="document.write('OnMouseDown');"
      onmouseup="document.write('OnMouseUp');"
      onclick="document.write('OnClick');">
<h1>Пример событий мыши</h1>
</body>
</html>
```

Прежде, чем перейти к следующему разделу, нужно сделать несколько замечаний. Во-первых, описанная выше последовательность событий мыши верна только для браузера Internet Explorer. В других браузерах последовательность событий и вообще логика работы механизма событий может отличаться. Во-вторых, на практике события `onmousedown` и `onmouseup` используются крайне редко, может, в каких-то игровых сценариях. В основном используется событие `onclick`, когда пользователь щелкнул на элементе веб-страницы. Данное событие можно определить отдельно для элемента, например для изображения:

```

```

8.7. Всплытие событий

При работе с событиями приходится иметь дело с явлением, которое называется всплытием событий. Чтобы понять, что это такое давайте рассмотрим листинг 8.2.

Листинг 8.2. Всплытие событий

```
<html>
<head>
<title>События</title>
</head>
<body onclick="window.alert('OnClick для Document');">
  <p onclick="window.alert('OnClick для
Paragraph');">Щелкните мышкой
    <span style="color: green" onclick="window.
      alert('OnClick для Span');">здесь</span>
  </p>
</body>
</html>
```

После загрузки страницы вы увидите надпись:

Щелкните мышкой здесь

При щелчке на "здесь" возникнет целая цепочка событий:

```
OnClick для Span
OnClick для Paragraph
OnClick для Document
```

Получается, что событие **onclick** передается последовательно родительскому элементу. Это явление и называется всплытием событий. Честно говоря, такое поведение не всегда желательно. Поэтому в JS предусмотрена возможность прерывания всплытия событий. Для этого предназначено свойство **cancelBubble** объекта **event**, кото-

рому нужно присвоить значение *true*. В некоторых браузерах для прерывания всплытия событий используется метод `stopPropogation()`. Рекомендуется использовать оба способа прерывания одновременно для совместимости с большим числом браузеров.

Пример прерывания всплытия событий приведен в листинге 8.3. Чтобы присвоить свойству `cancelBubble` значение *true* нам пришлось переписать обработчик события. Теперь в качестве обработчика используется наша функция `f_alert()`, которая устанавливает необходимое свойство и выводит переданное ей сообщение.

Листинг 8.3. Пример прерывания всплытия событий

```
<html>
<head>
<title>Проверка событий</title>

<script>
function f_alert(s, e) {
    e = e || window.event;
    if (e.stopPropogation) e.stopPropogation();
        else e.cancelBubble = true;
    window.alert(s);
}
</script>

</head>
<body onclick="f_alert('OnClick для Document', event);">
    <p onclick="f_alert('OnClick для Paragraph', event);">Щелкните мышкой
        <span style="color: green" onclick="f_alert('OnClick для Span', event);">здесь</span>
    </p>
</body>
</html>
```

В итоге, когда вы щелкните на надписи "здесь", вы получите только одно событие:

OnClick для Span

8.8. Действие по умолчанию

Для некоторых событий назначены действия по умолчанию, например, при нажатии кнопки **Submit** формы идет отправка содержимого формы на веб-сервер сценарию, который задан в свойствах формы. В некоторых ситуациях действия по умолчанию нужно отменить. Для этого нужно установить свойство **returnValue** в *false* или же использовать метод **preventDefault** (поддерживается не всеми браузерами):

```
function cancel(e) {  
    e = e || window.event;  
    if (e.preventDefault) e.preventDefault();  
    else e.returnValue = false;  
}
```

Далее для элемента, для которого нужно прервать действие по умолчанию, нужно вызвать эту функцию, например,

```

```

8.9. Обработчики событий

Как устанавливать обработчики событий, вы уже знаете. Для этого нужно задать событие и указать JS-код:

```
<тег событие="код">
```

Примеров было приведено уже предостаточно. Учитывая, что на практике не всегда нужно выполнять простые действия, заключа-

ющиеся из одного-двух операторов (например, вызов `window.alert`), рекомендуется создавать функции-обработчики событий, что мы и делали в этой главе. Этим вы убиваете двух зайцев: делаете ваш код проще для восприятий (а отсюда повышается читабельность, отлаживаемость кода и упрощается поиск ошибок) и позволяете преодолеть ограничение на максимальную длину значения HTML-атрибута (1024 символа).

Функции обработчики желательно определять в разделе `HEAD` - так будет логичнее для вас самих. Если же вы поместите код в `BODY` - не беда, браузер правильно обработает JS-код, но это будет не так нагляднее, как если бы обработчики были определены в `HEAD`. Например, в следующем сценарии (лист. 8.4) обработчик `onload` все равно будет вызван, хотя на самом деле он определен после тега `body`. Это происходит потому, что браузер сначала обрабатывает код сценария, а потом уже HTML-код.

Листинг 8.4. Пример обработчика события

```
<html>
<head>
    <title>Установка функции-обработчика</title>
</head>

<body onload="dialog();">
    <script>
        function dialog() {
            window.alert('Привет');
        }
    </script>
</body>

</html>
```

8.10. Объект event

В этой главе мы уже использовали объект **event**. Однако он заслуживает отдельного разговора, и мы его рассмотрим именно сейчас. Объект **event** используется для получения подробной информации о произошедшем событии и доступен он только в обработчиках событий (в других функциях и методах вы не можете его использовать). Когда наступает следующее событие, все значения свойства объекта **event** сбрасываются. Свойства объекта **event** описываются в таблице 8.5.

Таблица 8.5. Свойства объекта event

Свойство	Что содержит
srcElement	Ссылка на элемент-источник события.
currentTarget	Ссылка на элемент, в котором обрабатывается событие. Сылается на тот же элемент, что и ключевое слово this внутри обработчика события.
type	Позволяет узнать тип события (строка). Возвращается в нижнем регистре и без префикса on, например, для onclick свойство type равно "click"
clientX, clientY	Координаты (X, Y) события
screenX, screenY	Координаты (X, Y) события относительно окна
offsetX, offsetY	Координаты (X, Y) события относительно контейнера

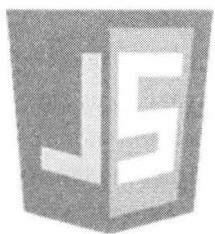
button	Число, указывающее, какая кнопка мыши была нажата: 0 - левая, 1 - средняя, 2 - правая
keyCode	Код нажатой на клавиатуре клавиши. В некоторых браузерах, например, в Firefox, данное свойство при обработке события onkeypress равно 0, а код символа доступен через свойство charCode. Если нажата функциональная клавиша, тогда charCode = 0, а код символа находится в keyCode.
altKey	Если это свойство равно <i>true</i> , то была нажата клавиша Alt вместе с другой клавишей, например, пользователь мог нажать Alt + A
ctrlKey	Если это свойство равно <i>true</i> , то была нажата клавиша Ctrl
shiftKey	Если <i>true</i> , то была нажата Shift
cancelBubble	Запрещает всплытие событий. Использование этого свойства было продемонстрировано ранее в этой главе.
returnValue	Определяет, будет ли выполняться действие по умолчанию. Использование этого свойства было продемонстрировано ранее в этой главе.
relatedTarget	Ссылка на элемент, с которого перешел курсор мыши.

В таблице 8.5 описаны не все возможные свойства объекта **event**. Наличие или отсутствие конкретного свойства зависит от уровня DOM.

DOM (Document Object Model) - объектная модель документа - это независимый от платформы интерфейс, позволяющий сценариям JavaScript получать доступ к содержимому HTML, XHTML и XML-документов, а также изменять содержимое, структуру и оформление этих документов. Существует четыре уровня DOM: DOM Level 0, DOM Level 1, DOM Level 2, DOM Level 3.

Стандарты DOM Level 0 и Level 1 так устарели, что заглядывать в них нет смысла. На смену Level 1 пришел Level 2, который внес много изменений в первые два уровня. В таблице 8.5 приведены свойства **event** согласно стандарта DOM Level 2.

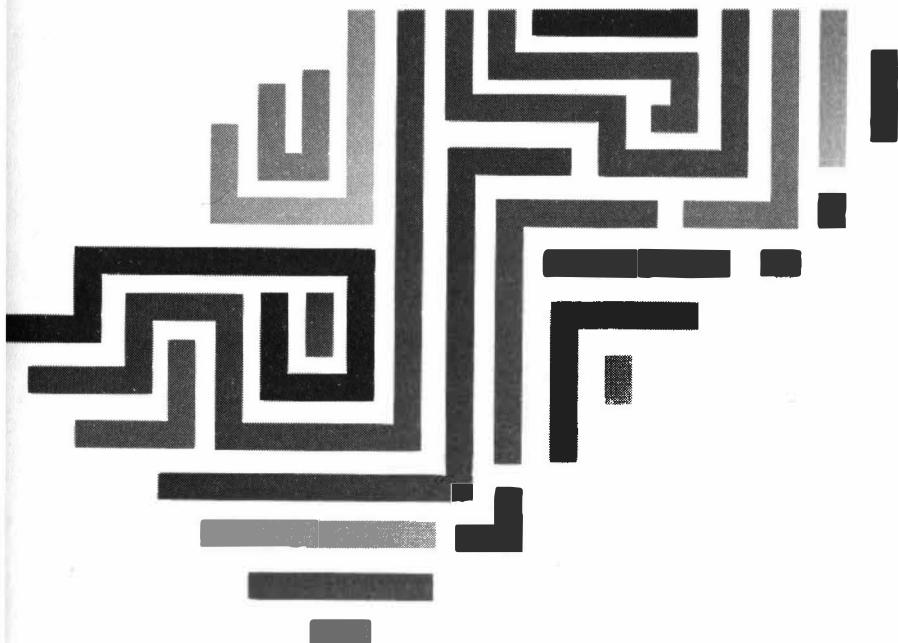
В следующей главе будет рассмотрена объектная модель браузера Internet Explorer. Почему именно этот браузер, думаю, ясно. Windows на данный момент все еще является самой популярной операционной системой, а в ней по умолчанию установлен именно Internet Explorer. В большинстве случаев весь описанный код будет работать и в других популярных браузерах (Chrome, Firefox, Opera), но более точный ответ, будет ли работать та или иная функция, можно получить в документации разработчика по конкретному браузеру. В подтверждение наших слов все скриншоты в следующей главе будут сделаны в браузере Google Chrome, чтобы продемонстрировать, что все написанное относится и к другим браузерам.



JavaScript

ГЛАВА 9.

Объектная модель браузера



9.1. Структура объектной модели

Объектной моделью браузера называется набор объектов, которые обеспечивают доступ к содержимому веб-страницы и некоторым функциям браузера.

Объектная модель представлена иерархией объектов. В ней есть объект верхнего уровня (родительский объект) и его дочерние объекты. У дочерних объектов могут быть собственные дочерние объекты и т.д. Доступ к таким объектам осуществляется через точку:

```
<родительский_объект>.〈дочерний_объект〉. {свойство}  
<родительский_объект>.〈дочерний_объект〉. {метод}
```

Часто, родительский объект (и даже некоторые дочерние объекты), можно не указывать. В качестве примера приведем метод `alert()` объекта `window`:

```
window.alert('Привет, читатель!');
```

Объект `window` является объектом самого верхнего уровня, представляющий веб-браузер, поэтому мы можем его не указывать, поскольку он подразумевается по умолчанию:

```
alert('Привет, читатель!');
```

Зато мы не раз опускали "window" при обращении к методу `document.write()`. Объект **document** является дочерним для объекта **window**, поэтому правильнее было бы писать код так:

```
window.document.write('Привет!');
```

Однако такая конструкция не очень удобна, поэтому мы использовали ее сокращенную форму (`document.write()`).

В этой главе будет рассмотрена объектная модель браузера Internet Explorer/Edge. Эта же модель полностью поддерживается самым популярным браузером - Google Chrome. Именно поэтому данная и многие другие книги по JS ориентируются на объектную модель IE.

9.2. Основные объекты объектной модели IE/Chrome

Кроме объекта **window** в объектной модели имеются следующие объекты (все они являются подчиненными объектами объекта `window`):

- **event** – используется для обработки событий.
- **frame** – используется для работы с фреймами. Сегодня использование фреймов не поощряется, поэтому этот объект мы рассматривать в книге не будем.
- **history** – позволяет обращаться к истории (журналу) браузера.
- **navigator** – предоставляет информацию о браузере.
- **location** – позволяет узнать и изменить URL-адрес текущей веб-страницы.
- **screen** – используется для доступа к характеристикам экрана компьютера пользователя.

- **document** – предоставляет доступ к документу, загруженному в браузер. Содержит следующие дочерние классы:
 - » **all** – предоставляет доступ ко всем элементам сразу.
 - » **anchors** – коллекция "якорей", заданных тегом <a>.
 - » **elements** – используется для доступа к элементам формы.
 - » **forms** – позволяет получить доступ к формам.
 - » **frames** – представляет все фреймы.
 - » **images** – предоставляет доступ ко всем изображениям.
 - » **links** – предоставляет доступ к ссылкам.
 - » **scripts** – предоставляет доступ к скриптам.
 - » **styleSheets** – предоставляет доступ к стилям.

9.3. Объект window

Как уже отмечалось, объект **window** представляет сам браузер. Далее мы рассмотрим свойства и методы этого объекта. Основные свойства объекта **window** приведены в таблице 9.1.

Таблица 9.1. Основные свойства объекта window

Свойство	Назначение
defaultStatus	Определяет сообщение, которое будет выводиться по умолчанию в строке состояния. Учтите, что не у каждого браузера есть строка состояния. Современные браузеры обходятся без этой строки и изменение данного свойства просто ни к чему не приведет

status	Сообщение, отображаемое в данный момент в строке состояния. В браузере IE есть строка состояния, но она может отсутствовать в других браузерах, например, ее нет в Google Chrome и любая попытка изменить установки свойства status просто ни к чему не приведет, как уже было отмечено ранее. <code>window.status = "Новый статус";</code>
length	Задает числом фреймов в данном окне. Как уже отмечалось, сегодня с фреймами уже никто не работает, поэтому и мы не будем
parent	Представляет собой ссылку на родительское окно
self	Позволяет сослаться на текущее окно
top	Является ссылкой на самое верхнее родительское окно
opener	Ссылка на окно, открывшее текущее окно
closed	Если равно <i>true</i> , то данное окно открыто, если <i>false</i> - закрыто (хотя по названию свойства можно подумать, что все наоборот)
screenLeft	Горизонтальная координата (X) левого верхнего угла окна. В Firefox вместо этого свойства используется свойство screenX
screenTop	Вертикальная координата (Y) левого верхнего угла окна. В Firefox вместо этого свойства используется свойство screenY
clientInformation	Объект navigator , то есть сам браузер

Для объекта **window** характерны события:

- **onload** – инициируется после загрузки веб-страницы
- **onunload** – происходит перед выгрузкой документа
- **onscroll** – инициируется при прокручивании содержимого окна или фрейма
- **onresize** – при изменении размеров окна
- **onblur** – происходит, когда окно теряет фокус
- **onfocus** – инициируется, когда окно получает фокус
- **onerror** – порождается, если в коде JavaScript возникает ошибка. В качестве обработчика этого события указывается функция, которой передается три параметра - описание ошибки, URL и номер строки. Функция должна вернуть *true*, если ошибка обработана и *false* в противном случае

Краткое описание методов объекта **window** приведено в таблице 9.2. Далее мы рассмотрим эти методы подробнее.

Таблица 9.2. Методы объекта window

Метод	Описание
alert()	Отображает диалоговое окно с пользовательским сообщением и кнопкой OK .
confirm()	Выдает окно подтверждения с кнопками OK и Отмена (см. гл. 1)
prompt()	Показывает окно с полем ввода и кнопками OK и Отмена (см. гл. 1)

showModalDialog()	Устаревший метод, который более не поддерживается. Для создания модальных окон теперь используется тег <dialog> и метод showModal()
open()	Открывает новое окно веб-браузера
close()	Позволяет закрыть текущее окно
blur()	Снимает фокус с окна
focus()	Переносит фокус на текущее окно, при этом генерируется событие onfocus
navigate(URL)	Загружает указанный URL в текущее окно браузера. Данный метод не поддерживается в Firefox
stop()	Позволяет прервать загрузку страницы. Не поддерживается браузером IE
resizeBy(x, y)	Позволяет изменить размеры окна
resizeTo(width, height)	Используется для установки новых размеров окна
moveBy(x, y)	Перемещает окно на x пикселей вправо, на y пикселей вниз
scrollBy(x, y)	Выполняет скроллинг окна на заданное расстояние
scrollTo(x, y)	Прокручивает окно в точку с заданными координатами x, y
setTimeout()	Устанавливает таймер. Мы рассмотрим пример использования этого метода

В таблице 9.2 приведены далеко не все методы объекта window, а только основные. Остальные методы будут рассмотрены по мере необходимости в этой главе.

9.3.1. Метод open(): создаем новые окна

Метод open() используется для открытия нового окна браузера и загрузки в него веб-страницы. Синтаксис следующий:

```
[var <окно> = ]window.open(<URL>, [<Имя окна>],  
                           [<Свойства >]);
```

Обязательным является только параметр URL, задающий адрес страницы, который нужно открыть в окне. Необязательный параметр Имя окна задает имя создаваемого окна, а параметр Свойства - свойства нового окна. При желании можно создать переменную окно, которую можно использовать для управления окном. Свойства окна (задаются последним) параметром приведены в таблице 9.3.

Таблица 9.3. Свойства нового окна (метод open())

Свойство	Описание
left, top	Используется для установки горизонтальной и вертикальной координаты левого верхнего угла нового окна
width, height	Ширина и высота создаваемого окна
fullscreen	Если это свойство равно yes или 1, тогда создаваемое окно будет открыто в полноэкранном режиме. Если свойство равно no или 0, режим окна будет обычным

resizable	Если это свойство равно yes или 1, тогда у нового окна будет возможность изменения размера. Если свойство равно по или 0, размер окна нельзя будет изменить
location	yes или 1 - у создаваемого окна будет строка адреса по или 0 - у создаваемого окна не будет адресной строки
menubar	yes или 1 - у создаваемого окна будет строка меню по или 0 - у создаваемого окна не будет меню
scrollbars	yes или 1 - у создаваемого окна будут полоски прокрутки по или 0 - у создаваемого окна не будет полосок прокрутки
status	yes или 1 - у создаваемого окна будет строка состояния по или 0 - у создаваемого окна не будет строки состояния
titlebar	yes или 1 - у создаваемого окна будет заголовок по или 0 - у создаваемого окна не будет заголовка
toolbar	yes или 1 - у создаваемого окна будет заголовок по или 0 - у создаваемого окна не будет заголовка

Пример использования метода `window.open()` приведен в листинге 9.1. При нажатии на кнопку **Открыть окно** происходит вызов функции `wopen()`, которая открывает окно с сайтом издательства и возвращает дескриптор окна. При вызове `window.open()` мы указываем URL, название сайта и строку, описывающую опции окна. В данном случае мы не будем отображать строку состояния и панель инструментов.

Листинг 9.1. Использование метода window.open()

```
<html>
<head>
    <title>Метод open</title>
    <script>

        function newwopen() {
            var options = "status = no,toolbar = no";
            return window.open("http://nit.center", "Сайт
издательства", options)
        }

    </script>
</head>
<body>
    <input type="button" onclick="newwopen();"
value="Открыть окно">
</body>
</html>
```

9.3.2. Отображение модального диалога

Ранее мы рассматривали методы alert(), prompt() и confirm(). Они были рассмотрены раньше времени, чтобы заинтересовать вас и продемонстрировать возможности JavaScript. Ранее был еще один важный метод - showModalDialog(), но он был заменен методом showModal(), применяемому к тегу **dialog**. Принцип следующий: тег <dialog> формирует модальное окно, а метод showModal() – отображает его.

Листинг 9.2 содержит код страницы, отображающей кнопку, при нажатии которой выводится диалог – форма со списком вариантом. Форма не просто отображается, а позволяет пользователю выбрать вариант списка и передает в сценарий выбранное значение.

Листинг 9.2. Модальное окно

```
<html>
<head>
<title>Модальное окно</title>
</head>
<body>

<!-- Диалог с формой --&gt;
&lt;dialog id="favDialog"&gt;
  &lt;form method="dialog"&gt;
    &lt;p&gt;&lt;label&gt;Любимая марка машины:&lt;/label&gt;
       &lt;select&gt;
         &lt;option&gt;&lt;/option&gt;
         &lt;option&gt;Audi&lt;/option&gt;
         &lt;option&gt;BMW&lt;/option&gt;
         &lt;option&gt;Opel&lt;/option&gt;
       &lt;/select&gt;
     &lt;/p&gt;
    &lt;menu&gt;
      &lt;button value="cancel"&gt;Отмена&lt;/button&gt;
      &lt;button id="confirmBtn" value="default"&gt;Выбрать&lt;/button&gt;
    &lt;/menu&gt;
  &lt;/form&gt;
&lt;/dialog&gt;

&lt;menu&gt;
  &lt;button id="chooseCar"&gt;Выбрать машину&lt;/button&gt;
&lt;/menu&gt;

&lt;output aria-live="polite"&gt;&lt;/output&gt;

&lt;script&gt;
var chooseButton = document.getElementById('chooseCar');
var favDialog = document.getElementById('favDialog');
var outputBox = document.querySelector('output');
var selectEl = document.querySelector('select');
var confirmBtn = document.getElementById('confirmBtn');</pre>
```

```
// открывает диалог favDialog
chooseButton.addEventListener('click', function onOpen()
{
    if (typeof favDialog.showModal === "function") {
        favDialog.showModal();
    } else {
        alert("Тег <dialog> не поддерживается браузером");
    }
});

selectEl.addEventListener('change', function onSelect(e)
{
    confirmBtn.value = selectEl.value;
});
// favDialog.addEventListener('close', function onClose() {
//     outputBox.value = favDialog.returnValue + " выбрано - "
+ (new Date()).toString();
});
</script>

</body>
</html>
```

На рис. 9.1 показан диалог и результат работы сценария.

Модальные диалоговые окна, в отличие от метода `prompt()`, очень удобно использовать, когда пользователю необходимо ввести несколько значений, например, для организации формы входа или же короткой анкеты регистрации.

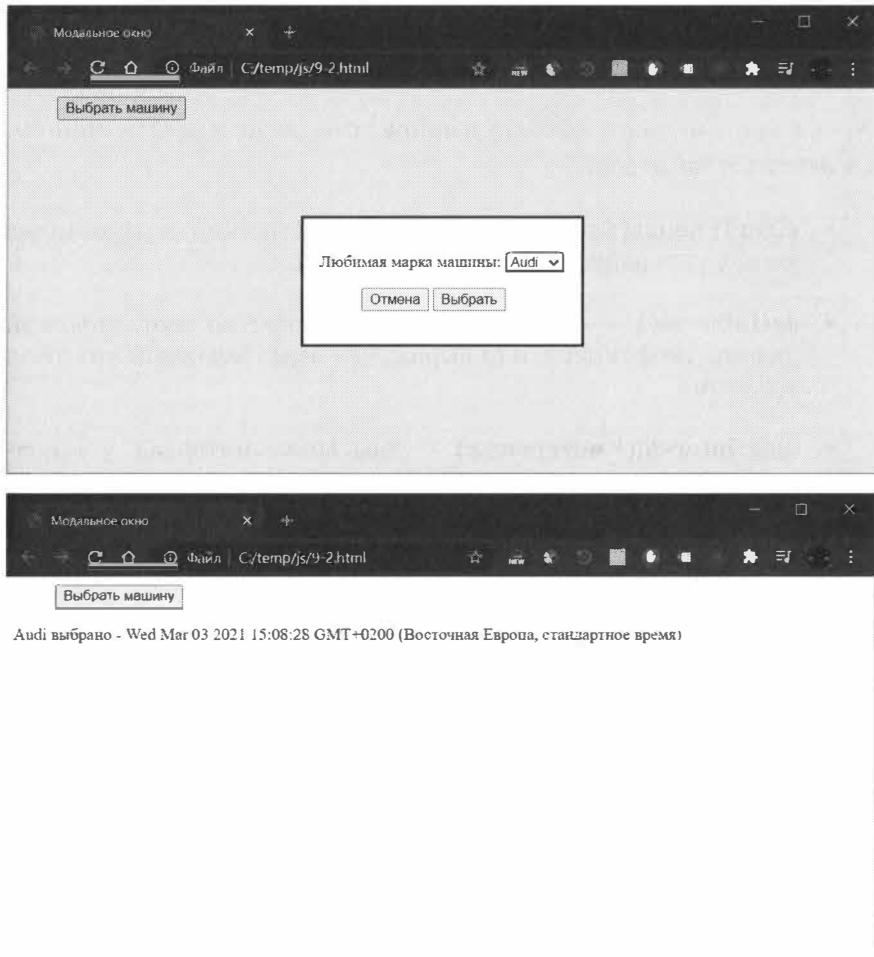


Рис. 9.1. Модальное окно

9.3.3. Метод setTimeout()

Метод `setTimeout()` используется для управления таймерами, а именно `setTimeout()` можно использовать для однократного выполнения определенной функции в заданное время. Вы задаете ин-

тервал времени, по истечению которого будет выполнена указанная функция.

Кроме этого метода у объекта **window** есть также и другие методы, касающиеся таймеров:

- **clearTimeout(<таймер>)** – сбрасывает таймер, который был установлен ранее методом setTimeout().
- **setInterval()** – создает таймер, многократно выполняющий указанную функцию или выражение через заданный интервал времени.
- **clearInterval(<интервал>)** – сбрасывает интервал, установленный методом setInterval().

Рассмотрим несколько примеров. Установить таймер можно так:

```
var timer = setTimeout(<функция или выражение>,
<инвервал>);
```

Сбросить таймер **timer** можно так:

```
clearTimeout(timer);
```

Метод setInterval() вызывается так:

```
var intr = setInterval(<функция или выражение>,
<инвервал>);
```

Как использовать таймеры? Первое, что приходит в голову - организовать часы на страничке. В листинге 9.3 функция Timerstart() запускает таймер посредством setInterval(). Каждую секунду (1000 мс) будет запускаться функция DisplayTime(), отображающая время. Для остановки таймера используется функция Timerstop().

Листинг 9.3. Пример использования таймеров

```
<html>
<head>
<title>Таймеры</title>
<script>

var clock;

function Timerstart() { // Запускает таймер
clock = setInterval("DisplayTime()", 1000);
}

function DisplayTime() { // Отображает время
var d = new Date();
var CurrTime = (d.getHours()<10) ? "0" : "";
CurrTime += d.getHours();
CurrTime += (d.getMinutes()<10) ? ":0" : ":";
CurrTime += d.getMinutes();
CurrTime += (d.getSeconds()<10) ? ":0" : ":";
CurrTime += d.getSeconds();
document.getElementById("div1").innerHTML = CurrTime;
}
function Timerstop() { // Останавливает таймер
clearInterval(clock);
}

</script>
</head>
<body>
<div id="div1"></div>

<input type="button" value="Start"
onclick="Timerstart();">
<input type="button" value="Stop" onclick="Timerstop();">

</body>
</html>
```



Рис. 9.2. Таймеры

9.4. Объект navigator: получение информации о браузере и системе

Объект **navigator** используется для получения информации о браузере. Свойства объекта **navigator** содержат много полезной информации:

- **appName** — содержит имя браузера;
- **appCodeName** — кодовое имя версии браузера;
- **appVersion** — позволяет узнать версию браузера;
- **appMinorVersion** — вторая цифра в номере версии браузера;
- **userAgent** — комбинация свойств appCodeName и appVersion;
- **cpuClass** — тип процессора компьютера пользователя;
- **platform** — название клиентской платформы;
- **systemLanguage** — содержит код языка операционной системы пользователя;
- **browserLanguage** — код языка браузера;
- **userLanguage** — код языка браузера;

- **onLine** — true, если клиент в настоящее время подключен к Интернету, и false, если мы - оффлайн;
- **cookieEnabled** — режим работы cookie: возвращает true, если прием cookie разрешен.

На практике данные свойства используются для получения информации о браузере. Рассмотрим пример:

```
console.log(navigator.userAgent);
```

в браузере Google Chrome (версии 88, текущей на момент написания данных строк) выведет следующую строку:

```
Mozilla/5.0 (Windows NT 10.0; Win64; x64)  
AppleWebKit/537.36 (KHTML, like Gecko)  
Chrome/88.0.4324.190 Safari/537.36
```

Информация о браузере может быть использована для отображения информации на языке клиента. Часто нужно, чтобы сайт определял язык клиента (например, русский или английский) и отображал страницы на языке, который выбран в качестве предпочтаемого в настройках браузера пользователя.

Используя свойства объекта **navigator**, вы можете попытаться определить язык браузера/системы и отобразить информацию на том языке. Если язык браузера/системы не поддерживается вашим сайтом, то информация будет отображена на языке по умолчанию, например, на русском. Это один из вариантов использования свойств объекта **navigator**.

Единственный недостаток - не все свойства поддерживаются некоторыми браузерами. Однако свойства вроде **AppName** и **userAgent** поддерживаются всегда, поэтому их можно использовать для опре-

деления браузера и написания более кросбраузерного кода, адаптированного под тот или иной браузер, например:

```
let userAgent = navigator.userAgent;
if (userAgent.indexOf("Explorer") != -1) {
    // код для IE
}
if (userAgent.indexOf("Firefox") != -1) {
    // код для Firefox
}
if (userAgent.indexOf("Opera") != -1) {
    // код для Opera
}
if (userAgent.indexOf("Chrome") != -1) {
    // код для IE
}
```

А вот пример определения мобильной/настольной версии браузера:

```
if (/Android|webOS|iPhone|iPad|iPod|BlackBerry|BB|Play
Book|IEMobile|Windows Phone|Kindle|Silk|Opera Mini/i.
test(navigator.userAgent)) {
    console.log('mobile');
}
else {
    console.log('desktop');
}
```

9.5. Объект **screen**: информация о мониторе пользователя

Объект **screen** позволяет получить информацию о мониторе пользователя. Рассмотрим некоторые свойства этого объекта:

- **width** — содержит ширину экрана в пикселях;

- **height** – содержит высоту экрана в пикселях;
- **availWidth** – ширина, доступная для окна браузера;
- **availHeight** – высота, доступная для окна браузера;
- **colorDepth** – глубина цвета: 4 – для 16 цветов, 8 – для 256 цветов, 32 – для 16,7 млн цветов.

9.6. Объект location: строка адреса пользователяского агента

Объект **location** предоставляет информацию об URL-адресе текущей страницы. Свойства объекта позволяют получить доступ к этой информации::

- **href** – содержит полный URL-адрес документа;
- **protocol** – содержит идентификатор протокола;
- **port** – позволяет узнать номер порта;
- **host** – имя сервера и номер порта;
- **hostname** – содержит имя узла;
- **pathname** – путь и имя файла;
- **search** – строка параметров, указанная после знака "?" (включая этот знак);
- **hash** – строка, указанная после знака "#" (включая этот знак).

Объект **location** обладает не только свойствами, но и методами:

- **assign()** – загружает указанный URL;
- **reload()** – перезагружает документ;
- **replace()** – загружает указанный в качестве параметра URL, при этом информация о предыдущем адресе удаляется из объекта **history**.

Нужно отметить, что загрузить новый документ можно не только с помощью метода **assign()**, но и с помощью изменения свойства **href**, например:

```
windows.location.href = "http://nit.center";
```

9.7. Объект **history**: список истории

Объект **history** предоставляет доступ к истории браузера, то есть к списку всех ранее просмотренных веб-страниц. Свойство **length** данного объекта содержит размер списка истории. Методы этого объекта следующие:

- **go(<номер>)** – переходит в списке истории на позицию с указанным номером.
- **back()** – загружает в окно браузера предыдущий документ, можно использовать для реализации кнопки **Назад**.
- **forward()** – загружает в окно браузера следующий документ из списка истории. Можно использовать для реализации кнопки **Вперед**.

Используя методы `back()` и `forward()`, можно создать кнопки или ссылки Назад/Вперед. Вот пример создания таких ссылок:

```
<p><a href="javascript:history.back();">Назад</a></p>
<p><a href="javascript:history.forward();">Вперед</a></p>
```

9.8. Объект `document`: доступ к элементам документа

Объект `document` предоставляет доступ к элементам документа. Данный объект содержит много всяких разных свойств, которые описаны в табл. 9.5.

Таблица 9.5. Свойства объекта `document`

Свойство	Назначение
<code>activeElement</code>	Содержит ссылку на активный элемент документа
<code>documentElement</code>	Содержит ссылку на тег <code><html></code>
<code>body</code>	Позволяет сослаться на все содержимое тега <code><body></code>
<code>title</code>	Содержит заголовок документа, то есть строку, указанную в теге <code><title></code>
<code>URL</code>	Содержит адрес документа
<code>referrer</code>	Адрес реферальной страницы, то есть страницы, с которой посетитель перешел на эту страницу
<code>parentWindow</code>	Ссылка на родительское окно
<code>cookie</code>	Предоставляет доступ к Cookies браузера

readyState	Позволяет определить состояние документа. Состояние может быть следующим: <i>uninitialized</i> - не инициализирован, недоступен <i>loading</i> - документ загружается <i>interactive</i> - загружен не полностью, но уже доступен для просмотра <i>complete</i> - полностью загружен
location	Объект location, рассмотренный ранее
selection	Объект selection, который будет рассмотрен чуть позже
fileCreatedDate	Дата создания файла документа (в виде строки)
fileModifiedDate	Дата последнего изменения
fileUpdatedDate	Дата обновления файла в кэше компьютера пользователя
lastModified	Дата и время последнего изменения документа
filesize	Размер файла
bgColor	Цвет фона документа
fgColor	Цвет текста страницы
linkColor	Цвет гиперссылок документа
alinkColor	Цвет активных ссылок
vlinkColor	Цвет посещенных ссылок

Объект document поддерживает следующие методы:

- **write(<текст>)** – добавляет в текущее место документа текст, заданный параметром;
- **writeln(<текст>)** – выводит в текущее место документа текст, заданный параметром, после чего добавляет символы возврата каретки и новой строки (\r\n). Поскольку эти символы полностью игнорируются браузером, поэтому результат будет такой же, что и у метода write();
- **getElementById(<ID элемента>)** – возвращает ссылку на элемент с указанным ID;
- **getElementByName(<название элемента>)** – возвращает ссылку на элемент по его имени;
- **elementFromPoint(<x>, <y>)** – возвращает ссылку на элемент, который находится по координатам <x>, <y>.

Каждый элемент на странице обладает собственными свойствами и методами. Таблица 9.6 содержит свойства, общие для всех элементов.

Таблица 9.6. Основные свойства элемента веб-страницы

Свойство	Назначение
all	Ссылается на коллекцию дочерних элементов
id	Ссылается на имя элемента, заданное параметром id
className	Ссылается на имя класса, заданное параметром class

sourceIndex	Содержит порядковый номер элемента, который можно использовать для ссылки на элемент из коллекции all
tagName	Имя тега элемента
parentElement	Ссылается на родительский элемент
length	Содержит число элементов в коллекции
height и width	Высота и ширина элемента
clientHeight и clientWidth	Высота и ширина элемента без учета рамок, границ и полосок прокрутки
clientLeft	Смещение левого края элемента относительно левого края родительского элемента без учета рамок, границ и т.д.
clientTop	Смещение верхнего края элемента относительно левого края родительского элемента без учета рамок, границ и т.д.
offsetHeight и offsetWidth	Высота и ширина элемента относительно родительского элемента
offsetLeft	Смещение левого края элемента относительно левого края родительского элемента
offsetParent	Ссылка на родительский элемент, относительно которого определяются свойства offsetHeight, offsetWidth, offsetLeft и offsetTop

innerText	Содержимое элемента без HTML-тегов. Если этому свойству присвоить новое значение, то изменится значение элемента
outerText	Содержимое элемента без HTML-тегов. Если присвоить свойству новое значение, то содержимое элемента заменится новым и будет изменен сам элемент
innerHTML	Содержимое элемента вместе с HTML-тегами. Если этому свойству присвоить новое значение, то изменится значение элемента
outerHTML	Содержимое элемента вместе с HTML-тегами. Если присвоить свойству новое значение, то содержимое
scrollHeight и scrollWidth	Высота и ширина содержимого элемента
scrollLeft и scrollTop	Положение горизонтальной и вертикальной полос прокрутки

В таблице 9.7 приведены методы, общие для всех элементов страницы.

Таблица 9.7. Основные методы элемента веб-страницы

Метод	Описание
<code>getAdjacentText (<местонахождение>)</code>	Возвращает текстовую строку в зависимости от указанного местонахождения
<code>insertAdjacentHTML (<местонахождение>, <текст>)</code>	Вставляет текст в место, заданное местонахождением

getAttribute (<имя параметра>, true false)	Возвращает значение параметра, который задан <именем параметра>. Если второй параметр равен <i>false</i> , то поиск параметра тега происходит без учета регистра символов
setAttribute (<имя параметра>, <значение>, true false)	Присваивает <Значение> параметру, который задан <именем параметра>. Если третий параметр равен <i>false</i> , то поиск параметра тега происходит без учета регистра символов
removeAttribute(<Имя параметра>, true false)	Удаляет параметр тега текущего элемента. Если второй параметр равен <i>false</i> , то поиск параметра тега происходит без учета регистра символов
clearAttributes()	Удаляет все параметра тега элемента, кроме параметров id и name
contains(<имя>)	Возвращает <i>true</i> , если элемент с этим именем содержится внутри текущего элемента

Используя все эти свойства и методы, вы можете манипулировать элементами страницы. Например, вы можете обратиться к любому изображению и получить его источник (заданный параметром тега **src**):

```
document.images[индекс].src
```

Нумерация, как обычно, начинается с 0. В цикле можно обойти все изображения (как и другие элементы страницы) с целью получения информации о них.

9.9. Объект style: доступ к таблице стилей

Объект **style** предоставляет доступ к каскадным таблицам стилей (CSS). Названия свойства этого объекта соответствуют атрибутам в CSS с небольшими отличиями, но удаляются символы " - ", при этом первые буквы всех слов в названии атрибута, кроме первого, делаются прописными. Например:

```
color = color  
font-size = fontSize
```

Рассмотрим пример (лист) изменения цвета текста в <div> с ID div1

Листинг 9.4. Изменение стиля с помощью JavaScript

```
<html>  
<body onload="OnLoad() ; ">  
<script>  
  
function OnLoad() {  
var div1 = document.getElementById("div1");  
div1.style.color = 'red';  
}  
  
</script>  
<div id="div1">Привет</div>  
</body>  
</html>
```

9.10. Объект selection: выделение текста

Объект **selection** позволяет управлять выделением фрагмента текста на странице. Если выбран текст, то свойство **type** объекта **selection** содержит значение *Text*, если ничего не выбрано, это свойство содержит значение *None*.

Методы объекта **selection**:

- **clear()** – очищает выделенный текст;
- **empty()** – снимает выделение текста.

Метод **getSelection()** объекта **window** возвращает объект **Selection**. У него есть различные свойства и методы. Самый часто используемый - метод **toString()**, возвращающий выделение в виде строки. Метод работает только в браузерах Chrome, Firefox и Opera. В IE код получения выделения нужно реализовывать через диапазоны. Рассмотрим сценарий 9.5, в котором приведен простой код получения выделения, который будет работать в браузерах Chrome, Firefox и Opera

Листинг 9.5. Получение выделения текста

```
<html>
<head>
    <title>Выделение</title>
</head>
<body>
<script>

function OnClick() {
    var selText = window.getSelection().toString();
    console.log(selText);

}
```

```
</script>
<div id="div1">Выделите текст и нажмите кнопку</div>
<p><input type="button" onclick="OnClick()" ;
value="Нажми меня">
</body>
</html>
```

Метод `getSelection()` получает выделение, которое затем преобразуется в текст методом `toString()` и отображаем методом `console.log`. Если же нужно обеспечить совместимость с браузером IE (а это нужно сделать, потому что все еще этот браузер занимает значительную долю на рынке браузеров), то код будет несколько сложнее. Нужно переписать функцию `OnClick()`, которая приведена в листинге 9.6. Остальной код будет таким же.

Примечание. Напоминаем, что метод `console.log` используется для вывода на консоль браузера. Для того, чтобы увидеть ее, нажмите F12 и перейдите на вкладку Консоль (Console).

Листинг 9.6. Универсальный код, работает в IE, Chrome, Firefox и Opera

```
function OnClick() {
    if (window.getSelection) {
        console.log(window.getSelection().toString());
    }
    else { // Для браузера IE
        if (document.selection.type=="Text") {
            let r1 = document.selection.createRange();
            console.log(r1.text);
            document.selection.empty();
        }
    else {
        console.log("Текст не выбран");
    }
}
```

```
} // IE block  
} //function
```

9.11. Немного практики. Работа с Cookies

9.11.1. Добавление сайта в Избранное

Многие сайты предоставляют возможность добавить сайт в избранное или установить его в качестве домашней страницы. Эти действия можно сделать и с помощью самого браузера, но наличие кнопок добавления в Избранное и/установки сайта в качестве домашней страницы позволит пользователю напомнить о том, что сайт можно добавить в закладки, особенно, если эти кнопки красиво оформлены. Установка сайта в качестве домашней страницы поможет несколько поднять трафик сайта – ведь при каждом запуске браузера вы получите обращение к своему сайту.

Чтобы добавить сайт в избранное, нужно использовать метод `addFavorite` объекта `external`. В качестве параметров этого методу нужно передать адрес страницы и ее описание. Рассмотрим код, реализующий добавление сайта в избранное. Вы можете использовать в качестве обработчика нажатия кнопки.

```
function addToFavorites() {  
    external.addFavorite("http://nit.center", "Nit.  
Center");  
    window.alert('Спасибо!');  
    return false;  
}  
...  
<p><a href="http://nit.center" onclick="return  
addFavorites();">  
Добавить в избранное</a><br>
```

9.11.2. Установка сайта в качестве домашней страницы

Чтобы установить ваш сайт в качестве домашней страницы, вы можете использовать функцию, код которой приведен ниже:

```
function setAsHomePage(obj) {  
    obj.style.behavior="url(#default#homepage)";  
    obj.setHomePage("http://nit.center");  
    window.alert('Спасибо!');  
    return false;  
}  
  
...  
<p><a href="http://nit.center" onclick="return  
setAsHomePage(this);">  
Сделать домашней страницей</a><br>
```

9.11.3. Работа с Cookies

Все современные браузеры позволяют хранить на компьютере пользователя небольшой объем информации, именуемый Cookies. Cookies обычно используются для хранения настроек пользователя, например, корзины, выбранного языка сайта, валюты и т.д.

У Cookies, конечно, есть свои достоинства и недостатки. Достоинства: вам не нужно хранить эту не очень важную информацию на своем сервере. Это снижает нагрузку на сервер и экономит дисковое пространство на нем.

А теперь о недостатках. Доступ к Cookies может получить любой другой скрипт, запущенный на компьютере пользователя (даже если он не устанавливал эти Cookies), поэтому в Cookies нельзя хранить конфиденциальную информацию, например, признак аутентификации, номера кредитных карточек, пароли и т.д. При переустановке

браузера или использовании другого браузера все данные, сохраненные в Cookies, также станут недоступными. Это второй недостаток.

Проверим, поддерживает ли браузер клиента Cookies:

```
if (navigator.cookieEnabled) {  
// Можем работать с Cookies  
}
```

Установить Cookies можно так:

```
document.cookie = "<Имя>=<Значение>; [expires=<Дата>;]  
[domain=<Имя домена>;] [path=<Путь>;] [secure;];";
```

В самом простом случае нужно указать пару <Имя>=<Значение>. В чуть более сложном - дату истечения срока действия. Дату нужно указывать только в таком формате:

```
Mon, 01 Mar 2021 00:00:01 GMT
```

По истечению этой даты Cookie будет удален. Получить дату в этом формате (чтоб не создавать ее вручную) можно с помощью метода `setTime()` и метода `toGMTString()`. Пример:

```
let dt = new Date();  
dt.setTime(dt.getTime() + 36000); // Время жизни 10 часов  
let Exp_Date = d.toGMTString(); // Конечная дата
```

Считать Cookie можно посредством обращения к `document.cookie`. Возвращенная строка будет содержать все установленные Cookie в формате "имя1=значение1; имя2=значение2".

Чтобы удалить Cookie, нужно установить его с истекшей датой. Другого способа, увы, нет.

Для облегчения работы с Cookies нами были разработаны функции, представленные в листинге 9.7. Используя эти функции, вы можете легко установить, прочитать и удалить Cookie.

Листинг 9.7. Функции для работы с Cookies

```
function getCookie(cname) {  
    var name = cname + "=";  
    var ca = document.cookie.split('');  
    for(var i = 0; i < ca.length; i++) {  
        var c = ca[i];  
        while (c.charAt(0) == ' ') {  
            c = c.substring(1);  
        }  
        if (c.indexOf(name) == 0) {  
            return c.substring(name.length, c.length);  
        }  
    }  
    return "";  
}  
  
function setCookie(name, value, opts = {}) {  
  
    /* opts = {  
        path: '/',  
        // при необходимости добавьте другие значения по  
        // умолчанию  
    }; */  
  
    if (opts.expires instanceof Date) {  
        opts.expires = opts.expires.toUTCString();  
    }  
  
    let updatedCookie = encodeURIComponent(name) + "=" +  
        encodeURIComponent(value);
```

```
for (let optionKey in opts) {  
    updatedCookie += "; " + optionKey;  
    let optionValue = opts[optionKey];  
    if (optionValue !== true) {  
        updatedCookie += "=" + optionValue;  
    }  
}  
  
document.cookie = updatedCookie;  
}
```

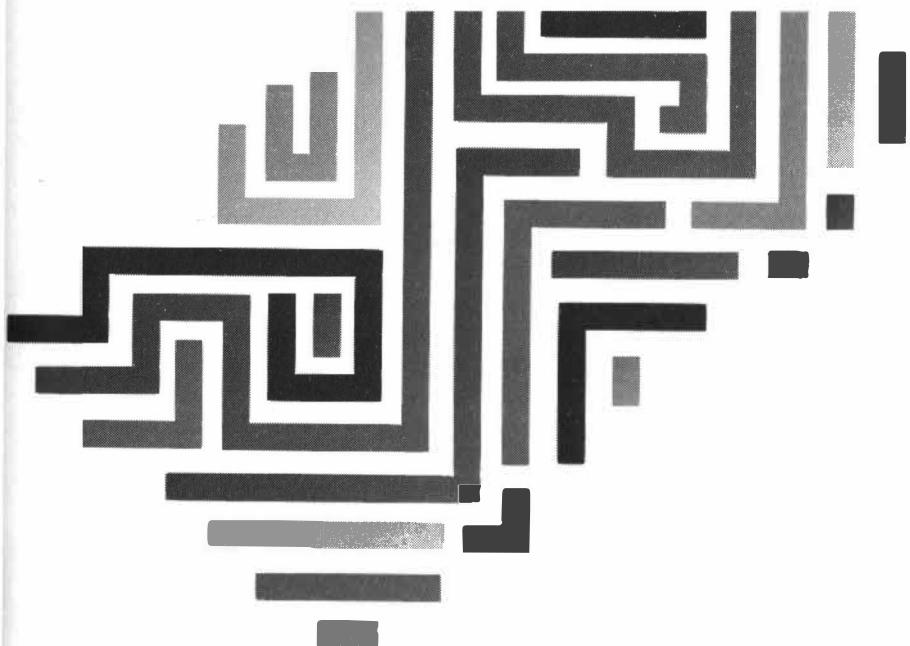
Использовать эти функции можно так:

```
// устанавливаем Cookies  
setCookie('saved_code', code, {secure: true, 'max-age': 60});  
setCookie('saved_card', card, {secure: true, 'max-age': 60});  
// читаем Cookie  
var saved_card = getCookie("saved_card");  
// Выводим Cookie  
window.alert(saved_card);
```

На этом сия глава заканчивается, а в следующей главе мы поговорим о том, как работать с формами в JavaScript.

ГЛАВА 10.

Работа с формами



10.1. Коллекция Forms

Представим, что в нашем документе есть форма `form1` с текстовым полем `firstname`. Обратиться к этому полю можно через коллекцию `forms`:

```
document.forms["form1"].firstname.value
```

При желании к форме можно обратиться и напрямую, что делает код компактнее:

```
document.form1.firstname.value
```

Существует и третий способ обращения к форме - по индексу:

```
document.forms[0].firstname.value
```

Нумерация форм начинается с 0. Если в документе только одна форма, то ее номер - 0.

Для получения доступа к элементу используется метод `getElementById()` объекта `document`:

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

Доступ к элементам формы также можно получить через коллекцию `elements`:

```
document.forms["loginform"].elements["password"].value  
document.forms["loginform"].elements[0].value  
document.forms[0].elements[0].value
```

Используйте тот метод обращения к элементам формы, который вам больше нравится.

10.2. Свойства, методы и события объекта формы

Объект формы обладает следующими свойствами:

- **action** – URL-адрес скрипта, обрабатывающего форму;
- **length** – число элементов в форме;
- **elements** – ссылка на коллекцию elements;
- **encoding** – MIME-тип передаваемых данных;
- **method** – способ передачи данных скрипту, указанному в action (get или post);
- **enctype** – метод кодирования данных формы, если нужно передать файл, то используется метод multipart/form-data;
- **name** – название формы;
- **target** – название фрейма, в который будет загружен документ.

Объект формы поддерживает всего два метода: submit() и reset(). Первый метод инициирует отправку формы на веб-сервер, второй - сбрасывает форму. Соответственно, есть два события - **onsubmit** и **onreset**. Первое происходит при отправке данных формы, второе - при сбросе формы.

10.3. Получение данных из поля ввода. Проверка правильности ввода

В этом разделе будут рассмотренные текстовое поле и поле для ввода пароля, которые создаются с помощью следующего HTML-кода:

```
<input type="text">  
<input type="password">
```

У этих элементов формы одинаковые свойства, методы и события. Сначала рассмотрим свойства (см. табл. 10.1)

Таблица 10.1. Свойства полей ввода текста и пароля

Свойство	Назначение
value	Значение элемента формы
defaultValue	Начальное значение, задаваемое параметром value
disabled	Позволяет выключить поле (при значении <i>true</i>)
form	Задает имя формы, к которой относится элемент ввода
maxLength	Максимальная длина поля, выражаемая в количестве символов, которые можно ввести в поле
name	Название элемента
readOnly	Если это свойство равно <i>true</i> , то поле нельзя редактировать, если <i>false</i> , то можно
type	Задает тип элемента формы: text – текстовое поле, password – поле для ввода пароля

Методы и связанные с ними события представлены в таблице 10.2.

Таблица 10.2. Методы и свойства полей ввода текста и пароля

Метод	Событие	Описание
blur()	onblur	Убирает фокус с текущего элемента формы. При потере фокуса происходит событие onblur
focus()	onfocus, onchange	Переводит фокус на текущий элемент формы. Когда определенный элемент формы получает фокус, происходит событие onfocus. Событие onchange происходит при получении изменения фокуса, изменении данных в поле или при отправке данных формы
select()	–	Выделяет текст в поле

10.4. Работа с `textarea`

В отличие от обычного поля ввода текста поле `textarea` позволяет вводить многострочный текст. Свойства, методы и события `textarea` такие же, как у поля ввода. Однако у `textarea` нет свойства `maxLength`, зато есть свойство `wrap`, позволяющее определить режим переноса слов. Это свойство может принимать следующее значение:

- **off** – перенос выключен.
- **physical** – физический перенос, то есть слова переносятся как на экране, так и при передаче данных.

- **virtual** – виртуальный перенос, то есть слова переносятся только на экране, но не при передаче на сервер.

Сейчас мы напишем простой сценарий, демонстрирующий работу с формой. Наша форма изображена на рис. 10.1. Форма содержит всего три элемента - поле ввода, **textarea** и кнопка. При нажатии кнопки **Добавить слово** введенное в текстовое поле слово будет добавлено в **textarea**. Листинг 10.1 содержит комментарии, прочитайте их для лучшего понимания программы.

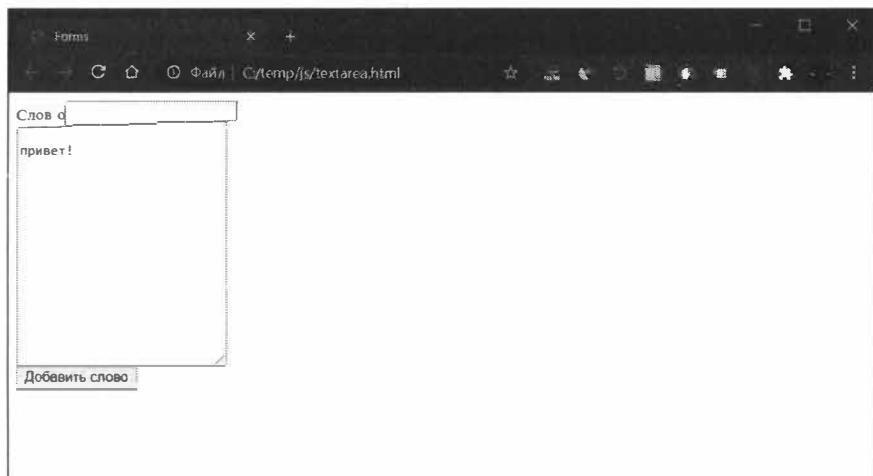


Рис. 10.1. Наша форма

Листинг 10.1. Работа с текстовыми полями

```
<html>
  <head>
    <title>Forms</title>
  </head>
<body>
  <script>
    function AddWord() {
      // получаем значение текстового поля
```

```
var text1 = document.form1.text1.value;

// если текст не введен, выходим
if (text1 == "") { windows.alert('Введите текст');
return ""; }

// получаем значение textarea
var tal = document.form1.tal.value;
var result = tal + "\n" + text1;

// новое значение textarea
document.form1.tal.value = result;

// очищаем текстовое поле
document.form1.text1.value = "";
return text1;      // возвращает введенный текст

}

</script>
<form name="form1">
    Слово <input type="text" name="text1" id="text1"><br>
    <textarea name="tal" id="tal" cols="30" rows="10"></
    textarea><br>
    <input type="button" value="Добавить в список"
    onclick="AddWord(); "><br>
</form>
</body>
</html>
```

10.5. Работа с флагками

У флагков и переключателей несколько другой набор свойств, методов и событий. Как обычно, сначала рассмотрим свойства (таблица 10.3).

Таблица 10.3. Свойства флагков и переключателей

Свойство	Описание
checked	Если свойство равно <i>true</i> , переключатель или флагок находится во включенном состоянии
value	Содержит значение текущего элемента формы.
defaultChecked	Позволяет узнать, включен ли флагок или переключатель по умолчанию (<i>true</i> - да, <i>false</i> - нет)
disabled	Если <i>true</i> , то элемент формы выключен (его нельзя изменять)
indeterminate	Флагок находится в неопределенном состоянии (закрашен серым цветом). Возвращает <i>false</i> или <i>true</i>
form	Ссылка на форму, в которой находится элемент
name	Имя элемента
type	Тип элемента формы

Из методов флагки и переключатели поддерживают только `blue()` и `focus()`, которые были рассмотрены раньше. События тоже аналогичны рассмотренным ранее: `onblur()`, `onfocus()`. Также есть событие `onclick()`, которое возникает при выборе элемента.

Теперь рассмотрим небольшой пример. Пусть у нас есть группа переключателей `rg1`:

```
<input type="radio" name="rg1" id="radio1" value="1"
checked>Вариант 1
<input type="radio" name="rg1" id="radio2"
value="2">Вариант 2<br>
<input type="radio" name="rg1" id="radio3"
value="3">Вариант 3<br>
```

Обойти группу переключателей в цикле можно так:

```
let counter = document.form1.rg1.length;
for (let i=0; i < counter; i++) {
    if (document.form.rg1.item(i).checked) {
        console.log(document.form.rg1.item(i).value);
        break;
    }
}
```

После того, как мы находим включенный переключатель, мы выводим его значение (задается атрибутом **value**) на консоль браузера и прерываем цикл - остальные переключатели нет смысла просматривать, так как выбранным в группе может быть только один переключатель (не путать с флагжком - checkbox!).

10.6. Работа с кнопками

Свойства кнопки подобны свойствам других элементов формы:

- **value** – значение кнопки (текст, который отображается на ней)
- **disabled** – если это свойство равно true, кнопка будет неактивной;
- **form** – ссылка на форму, в которой находится кнопка;
- **name** – название кнопки;
- **type** – тип элемента формы (button) .

Методы тоже уже вам знакомы: blur() и focus(). События аналогичны переключателям: onblur(), onclick() и onfocus(). Далее приведен пример работы с кнопками. Наша форма будет содержать две кнопки – **Нажми меня** и **Проверка**. При нажатии на **Нажми меня** происходит инвертирование кнопки **Проверка**. То есть при одном нажатии кнопка **Test** будет выключена, при другом - включена и т.д.

Листинг 10.2. Пример работы с кнопками

```
<html>
  <head>
    <title>Формы</title>
  <script>
    function OnBtnClick() {
      document.form1.button2.disabled = !document.form1.
button2.disabled;
    }
  </script>

  </head>
  <body>

    <form name="form1">
      <input type="button" name="button1" id="button1"
value="Кнопка 1" onclick="OnBtnClick();"><br>
      <input type="button" name="button2" id="button2"
value="Кнопка 2"><br>
    </form>
  </body>
</html>
```

10.7. Проверка правильности e-mail

Очень часто перед отправкой формы на сервер требуется проверить введенные данные, например, ввел ли пользователь имя, e-mail, установил ли пароль, совпадают ли пароли (если речь идет о форме регистрации). Учитывая все полученные знания, вы сами можете написать код такой проверки. Сложность может вызвать разве что проверка корректности электронного адреса. Если проверять корректность e-mail с помощью обычных проверок, то получите некомпактный и неэффективный код, перегруженный множеством опера-

торов `if` и не факт, что он будет работать правильно. Например, все мы знаем, что в любом электронном адресе должны быть символы "@" и ".", но если проверить наличие в строке только этих символов, то проверка явно не будет полной. Например:

```
@us.er@  
.user@  
.user@domain.  
.u@d.@
```

Все эти адреса вряд ли можно назвать корректными адресами. Какие есть мысли? Проверять, чтобы в строке был только один символ @. Но все равно есть много неправильных вариантов, например

```
us.r@domain  
пользователь@domain.ru  
user@domain...
```

Чтобы не загромождать код лишними проверками есть один выход - использовать регулярные выражения. Только с помощью регулярных выражений можно быстро и компактно проверить правильность e-mail. Давайте напишем функцию `validateEmail()`, которая будет проверять правильность введенного e-mail:

```
function validateEmail(email) {  
    var re = /^(([^<>()[]\\.,;:\\s@\\"]+\\.\\[^<>()  
[]\\.,;:\\s@\\"]+)*|(\".+\\\"))@((\\[[0-9]{1,3}\\.[0-9]  
{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\])|(([a-zA-Z\\-0-9]+\\.)+[a-  
Z]'{2,}))$/;  
    return re.test(email);  
}
```

Да, регулярное выражение очень сложное, но оно учитывает все возможные варианты правильного построения e-mail и избавляет вас от множества ненужных проверок.

В листинге 10.3 приведен полный код сценария проверки e-mail:

Листинг 10.3. Проверка допустимости e-mail

```
<html>
<head>
<title>Формы</title>
<script>

function validateEmail(email) {
var re = /^[^<>()[]\.\,\,;\:\s@"]+(\.[^<>()
[]\.\,\,;\:\s@"]+)*|(\".+\")@\((\[[0-9]{1,3}\.[0-9]
{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\))|(([a-zA-Z\-\-0-9]+\.)+[a-
-zA-Z]{2,}))$/;
return re.test(email);
}

function checkMail() {

var email = document.form1.email.value;

if (validateEmail(email)) window.alert("OK");
else window.alert("Email неправильный");

}

</script>
<body>

<form name="form1">
E-mail: <input type="text" name="email" id="email">
      <input type="button" value="Проверка"
onclick="checkMail();"
      >
</form>

</body>
</html>
```

В функции `check()` мы передаем содержимое текстового поля `email` нашей функции `validateEmail()`. Если она возвращает `true`, мы выво-

дим сообщение **OK**, в противном случае мы сообщаем, что e-mail не-корректен.

Теперь давайте усложним задачу. Наверняка вы видели в Интернете сценарии, где проверка ввода осуществляется при самом вводе, а не при нажатии кнопки **Проверка**. Такие кнопки уже давно в прошлом и в современных сценариях будут выглядеть архаично. Сейчас мы перепишем наш сценарий так, чтобы он проверял правильность e-mail на лету и выводил в определенный элемент документа результат проверки. Первым делом нужно переписать саму функцию `check()`. Нам уже не нужно выводить сообщения с помощью `window.alert()`, поэтому результат проверки мы будем выводить в отдельный `` документа. Вот измененный код:

```
function check() {  
  
    var email = document.form1.email.value;  
    if (validateEmail(email)) document.  
        getElementById("span1").innerText = "OK";  
    else document.getElementById("span1").innerText = "Email  
неправильный";  
  
}
```

Логика функции осталась той же, изменился только способ отображения результата. Все остальное - дело техники. Для проверки "на лету" нужно использовать событие `onKeyDown` нашего поля ввода:

```
<input type="text" name="email" id="email"  
onKeyDown="check();">
```

В качестве обработчика события мы используем нашу функцию `check()`. Собственно, в нашей форме уже нет необходимости в кнопке **Проверка**. Наша форма теперь изображена на рис. 10.2. Как видите, форма содержит только поле ввода e-mail. Элемент, в который выводится результат обработки, может находиться за пределами формы.



Рис. 10.2. Форма ввода

Измененный сценарий представлен в листинге 10.4.

Листинг 10.4. Измененный сценарий

```
<html>
<head>
<title>Формы</title>

<script>

function validateEmail(email) {
    var re = /^[^<>() \\\\.,,:\\s@\\"]+(\\.\\[^<>()
[\\]\\\\\.,,:\\s@\\"]+)*|(\\".+\\")@((\\[[0-9]{1,3}\\.[0-9]
{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\])|(([a-zA-Z\\-0-9]+\\.)+[a-
-zA-Z]{2,}))$/;
return re.test(email);
}

function check() {

var email = document.form1.email.value;
if (validateEmail(email)) document.
getElementById("span1").innerText = "OK";
else document.getElementById("span1").innerText = "Email
неправильный";
}

</script>
<body>
```

```
<form name="form1">E-mail: <input type="text" name="email" id="email" onKeyDown="check();">
</form>

<span id="span1"></span>

</body>
</html>
```

Аналогичным образом, используя событие `onKeyDown`, можно проверить правильность других полей формы, например, указал ли пользователь свой номер телефона. Ниже, кстати, представлена функция для проверки номера телефона в международном формате (+код_страны(код_оператора)xxx-xx-xx):

```
function isValidPhone(Phone) {
    return /^+\d{1,2} \(\d{3}\) \d{3}-\d{2}-\d{2}$/.test(Phone);
}
```

Обратите внимание, что функции проверки e-mail и номера телефона проверяют, соответствуют ли введенные пользователем данные определенному шаблону. Они не проверяют существование такого e-mail и такого номера телефона. Например, вы можете ввести +33(011)010-00-00. Это корректный номер телефона, то есть такой номер может существовать, но существует ли - никто не знает (лично я не проверял). Проверить существование e-mail можно только путем отправки сообщения на этот адрес, но, как правило, этим никто не занимается. С технической точки зрения реализовать такую проверку можно: отправить e-mail, получить ответ от сервера и выдать результат. Однако такие проверки порождают ненужный трафик.

Также на некоторых сайтах есть возможность проверки не только корректности введенных данных, но и их допустимости. Например, вы при регистрации ввели e-mail user@domain.com. С точки зрения функции проверки он является корректным, поскольку соответству-

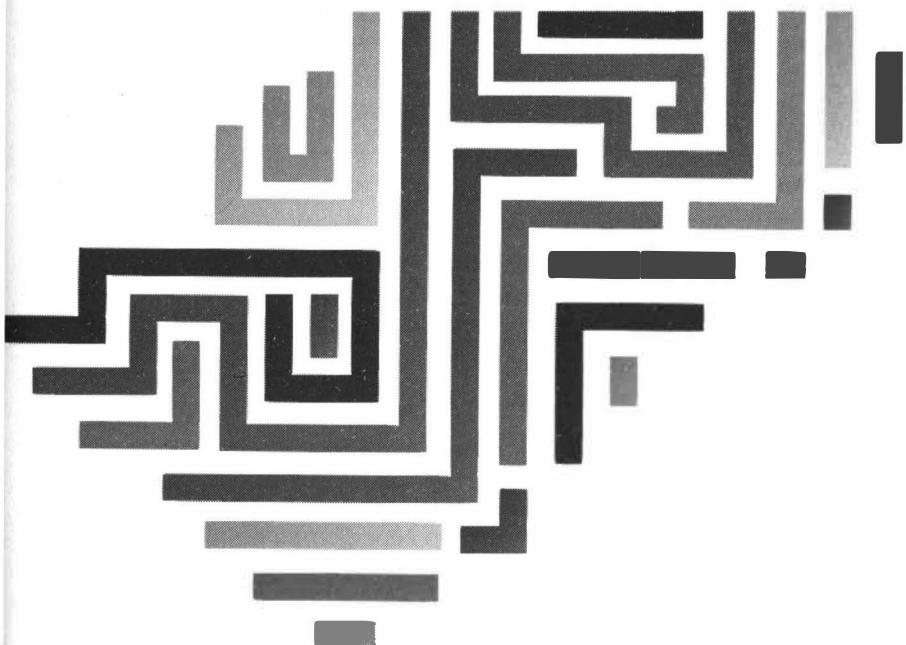
ет регулярному выражению. Однако этот e-mail может принадлежать другому пользователю (кто-то уже указывал его при регистрации). К сожалению, JavaScript не умеет непосредственно обращаться к базе данных, чтобы проверить, есть ли такой e-mail в БД. Поэтому у вас есть два варианта:

- Остановиться на функции `validateEmail()` и отправлять данные на сервер в случае, если они похожи на правильные. Дальнейшей обработкой будет заниматься программа на сервере (как правило, это будет PHP-сценарий).
- Освоить технологию AJAX, позволяющая вызывать стороннюю программу для обработки данных. Например, наш сценарий вызовет PHP-сценарий, который проверит, указывался ли такой e-mail при регистрации другого пользователя, и передаст результат проверки нашему JS-сценарию. Наш сценарий на основании полученного ответа выполнит определенные действия (сообщит пользователю, что e-mail занят или свободен).

Теперь вы знаете достаточно, чтобы приступить к изучению библиотек JavaScript. Далее мы вкратце рассмотрим одни из самых популярных – jQuery и jQuery UI.

ГЛАВА 11.

Библиотека jQuery



Несмотря на то, что "ванильный" JavaScript в современном виде позволяет сделать многие вещи из библиотеки jQuery без привлечения самой библиотеки, jQuery является по-прежнему одной из самых распространенных библиотек (по некоторым данным до 70% сайтов в интернете используют данную библиотеку), зачастую существенно упрощая разработку. Именно поэтому мы включили ее в справочник.

11.1. Подключение библиотеки jQuery

jQuery - это библиотека JavaScript, позволяющая улучшить взаимодействие HTML и JavaScript. Она существенно облегчает доступ к атрибутам и содержимому элементов DOM, предоставляет удобный API для работы с AJAX, а также является основой для надстройки jQuery UI, которая часто применяется при проектировании пользовательских интерфейсов.

О том, как подключить jQuery, написано на сайте <http://jquery.com/download/>. Это официальный сайт jQuery и строго не рекомендуется загружать библиотеку со сторонних сайтов - ведь ее код может содержать уязвимости, внесенные недоброжелателями.

На момент написания книги актуальной является версия 3.6.0. Загрузить сжатую продакшн-версию можно по адресу:

<https://code.jquery.com/jquery-3.6.0.min.js>

Если вы хотите изучать код библиотеки и вам интересно, как все устроено, можно загрузить несжатую версию для разработчика:

<https://code.jquery.com/jquery-3.6.0.js>

В продакшене (когда будете сдавать сайт заказчику) рекомендуется использовать сжатую версию:

- Она занимает меньше места, следовательно, странички, к которым она подключена, будут загружаться быстрее.
- Всякие инструменты SEO-оптимизации будут "ругаться" на несжатую версию. Есть даже основания считать, что несжатые версии JavaScript-сценариев плохо влияют на поисковую оптимизацию. Но, по сути, влияют они только косвенно: поисковые машины оценивают скорость загрузки странички, а чем меньше размер подключенных к ней сценариев, тем быстрее она загружается.

Поместите загруженный файл в подкаталог **js** вашего приложения (сайта). Подключить jQuery можно, как и любую другую библиотеку:

```
<head>
...
<script src="js/jquery-3.6.0.min.js"></script>
</head>
```

При желании можно и не загружать библиотеку на свой сайт, а грузить ее с code.jquery.com:

```
<head>
...
<script src="https://code.jquery.com/jquery-3.6.0.min.
js"></script>
</head>
```

Однако рекомендуется все-таки загрузить ее на свой сайт - с вашего сайта она будет загружаться быстрее.

Также рекомендуется подключать скрипты не в **<head>** (хотя это не запрещается и ранее даже рекомендовалось), а до тега **</body>** - в результате странички будут загружаться быстрее, поскольку сначала загрузится контент, а уже потом скрипты, которые его обрабатывают.

11.2. Выбор элемента HTML. Селекторы

jQuery позволяет гибко выбирать элементы страницы. Можно, например, выбрать какой-то определенный элемент, все элементы, все элементы определенного типа, например, поля ввода и т.д.

После того, как элемент, выбран, с ним можно работать. В большинстве случаев воздействие на элемент заключается в изменении его атрибутов. Это мы тоже рассмотрим, но позже. Нужно отметить, что в этой книге мы не сможем рассмотреть все возможности jQuery – остановимся только на основных. Если вы заинтересовались, в Сети вы без проблем найдете дополнительную информацию об этой библиотеке.

11.2.1. Выбор всех элементов

Для выбора всех элементов используется селектор "*". Пример:

```
<script>
$(function() {
    alert($(".*").length);
});
</script>
```

Но абсолютно все элементы нам не нужны, например, зачем нам скрипты из `<head>`? Чтобы выбрать все элементы из секции `<body>`, нужно изменить наш код так:

```
$(function() {
    alert($(".*", document.body).length);
});
```

Далее мы рассмотрим селекторы, позволяющие организовать более точный выбор элементов.

11.2.2. Выбор элемента по его id

Представим, что у нас есть <div> с id="div1". Нам нужно изменить его рамку, рамка должна быть толщиной 3 пикселя, состоять из сплошной линии и быть окрашенной в красный цвет. Полный код сценарий приведен в листинге 11.1, а результат его работы - на рис. 11.1.

Листинг 11.1. Поиск элемента страницы по его id

```
<html>
<head>
    <title>Выбор по id</title>
</head>
<body>
    <div id="divInfo">Делаем синюю рамку вокруг текста</div></body>
    <script src="https://code.jquery.com/jquery-3.6.0.js"></script>
    <script>
        $(function() {
            $("#divInfo").css("border", "3px solid #00f");
        });
    </script>
</html>
```

Обратите внимание как все просто: вместо * указываем имя нужного нам элемента, а дальше с помощью метода css() изменяем его внешний вид. Первый параметр метода css() указывает атрибут ("border"), второй - значение ("3px solid #00f").

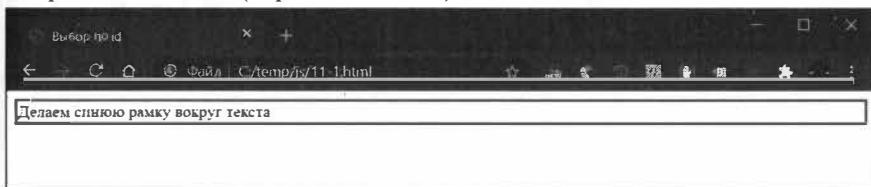


Рис. 11.1. Результат работы сценария из лист. 11.1

11.2.3. Поиск всех элементов определенного типа

Теперь рассмотрим следующий пример. На этот раз нам нужно отыскать все элементы определенного типа, например, все `<div>` и установить для них одинаковые параметры. У нас будут два тега `<div>` с разными значениями `id` (чтобы быть ближе к реальным условиям, где у каждого элемента будет свое имя или вообще не будет имени), мы найдем все `div` и установим для них зеленую рамку (лист. 11.2). Также мы устанавливаем свойство `margin` для всех `div` в документе.

Листинг 11.2. Выбор всех элементов определенного типа

```
<html>
<head>
    <title>Выбор элемента по типу</title>
</head>
<body>
    <div id="myDiv">Блок 1</div>
    <div id="someDiv">Блок 2</div>
    <script src="https://code.jquery.com/jquery-
3.6.0.js"></script>
    <script >
        $(function() {
            $("div").css("border", "3px solid green");
        });
    </script >
</body>
</html>
```

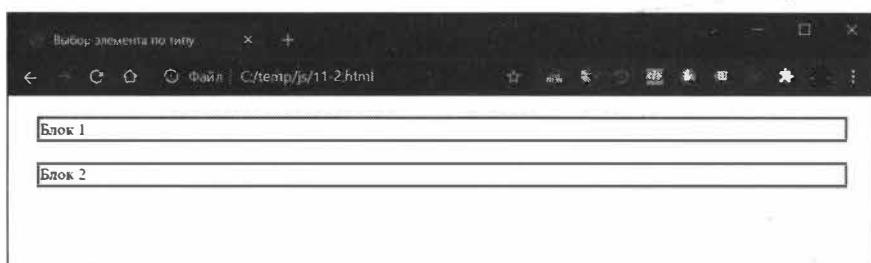


Рис. 11.2. Результат работы сценария из лист. 11.2

```
$("div").css("margin", "20px");  
});  
</script>  
  
</body>  
</html>
```

11.2.4. Поиск элементов по имени класса

Чтобы найти элементы по имени класса (не по **id**), нужно задать имя класса, указав перед ним точку:

```
$(".text").css("border", "1px solid #f00");
```

Напомним, что в HTML-коде имя класса задается с помощью атрибута **class**:

```
<p class="text">Текст</p>
```

В селекторе вы можете указать через запятую не только один элемент, а несколько. Например, следующий селектор выбирает все элементы **<div>** и весь класс **text**, относящийся только к тегу **<p>**:

```
$( "div, p.text" ).css("border", "3px solid #f00");
```

Если в вашем коде будут еще элементы, принадлежащие к классу **text**, отличные от **<p>**, они не будут выделены, следовательно, стиль CSS не будет к ним применен. Посмотрите листинг 11.3. В нем к классу **text** принадлежат абзац и список. На рис. 11.3 показан результат выполнения нашего сценария - список не был выделен.

Листинг 11.3. Пример выбора нескольких объектов

```
<html>  
<head>  
  <title>Пример выбора нескольких объектов</title>
```

```

</head>
<body>
  <div id="myDiv">Блок 1</div>
  <div id="someDiv">Блок 2</div>
  <p class="text">Текст</p>
  <ul class="text"><li>Элемент 1</li></ul>
  <script src="https://code.jquery.com/jquery-
3.6.0.js"></script>
  <script>
    $(function() {
      $("div, p.text").css("border", "4px solid red");
    });
  </script>
</body>
</html>

```

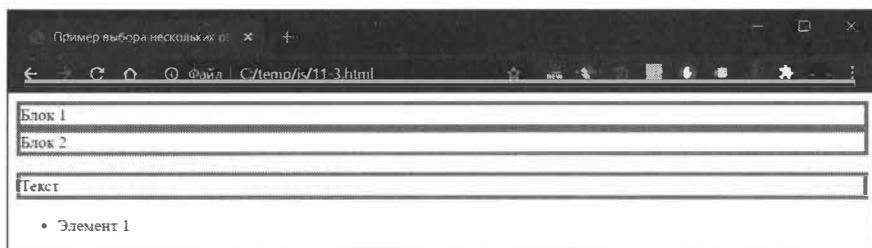


Рис. 11.3. Выбор нескольких разных элементов

11.2.5. Селекторы + и ~

Также вы можете использовать селекторы + и ~. Селектор + выбирает элемент, следующий непосредственно за указанным (известным элементом). Например:

предыдущий + следующий

Селектор ~ позволяет найти все элементы, которые находятся на одном уровне с указанным элементом, то есть сестринские (не дочерние) элементы.

11.3. Фильтры

Использование фильтров позволяет очень легко выбрать те или иные элементы страницы, например, выбрать первый элемент или выбрать четные элементы. Список фильтров:

- **:first** – позволяет выбрать первый элемент
- **:last** – используется для выбора последнего элемента
- **:even** – используется для выбора четных элементов.
- **:odd** – выбирает нечетные элементы
- **:eq(n)** – позволяет выбрать элемент с индексом n
- **:gt(n)** – выбирает все элементы с индексом > n
- **:lt(n)** – позволяет выбрать все элементы с индексом < n
- **:not(фильтр)** – используется для обращения выбора фильтра, указанного в скобках
- **:header** – выбирает все элементы, являющиеся заголовками
- **:contains(строка)** – выбирает все элементы, содержащие строку
- **:empty** – выбирает все пустые элементы
- **:checked** – выбирает выбранные элементы checkbox
- **:selected** – выбирает выбранные элементы списка

Далее мы рассмотрим все эти фильтры на практических примерах.

11.3.1. Фильтр :first

Представим, что у нас есть следующая таблица:

```
<table style="width: 100%">
  <tr>
    <td>1</td><td>2</td><td>3</td><td>4</td>
  </tr>
  <tr>
    <td>5</td><td>6</td><td>7</td><td>8</td>
  </tr>
  <tr>
    <td>9</td><td>10</td><td>11</td><td>12</td>
  </tr>
  <tr>
    <td>13</td><td>14</td><td>15</td><td>16</td>
  </tr>
</table>
```

Выглядит таблица ничем не примечательно: нет ни рамки, ни фона. Только и того, что она растянута на всю ширину окна.

Давайте попробуем изменить цвет фона первого ряда (tr). Для этого используется фильтр :first. Вот как его нужно использовать:

```
$( "tr:first" ).css( "background-color", "lightgray" );
```

Полный код страницы приведен в листинге 11.4, а сама страница изображена на рис. 11.4.

Листинг 11.4. Пример использования фильтра :first

```
<html>
<head>
  <title>Пример использования фильтра first</title>
</head>
<body>
<table style="width: 100%">
```

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Рис. 11.4. Наша таблица

```
<tr>
    <td>1</td><td>2</td><td>3</td><td>4</td>
</tr>
<tr>
    <td>5</td><td>6</td><td>7</td><td>8</td>
</tr>
<tr>
    <td>9</td><td>10</td><td>11</td><td>12</td>
</tr>
<tr>
    <td>13</td><td>14</td><td>15</td><td>16</td>
</tr>
</table>
<script src="https://code.jquery.com/jquery-
3.6.0.js"></script>
<script>
$(function() {
    $("tr:first").css("background-color", "lightgray");
});
</script>

</body>
</html>
```

11.3.2. Выбор последнего ряда таблицы

Аналогично, для выбора последнего элемента используется фильтр :last. Выбрать четные элементы можно фильтром :even, а нечетных - :odd:

```
$(“tr:last”).css(“background-color”, “red”);
$(“tr:even”).css(“background-color”, “green”);
$(“tr:odd”).css(“background-color”, “yellow”);
```

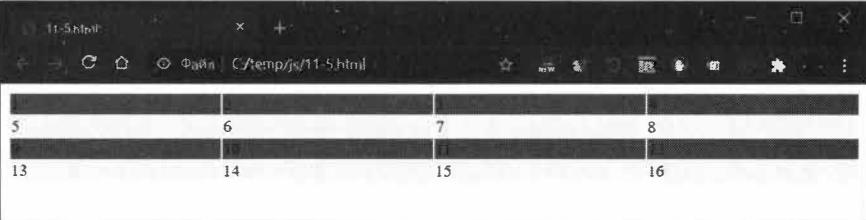
11.3.3. Выбор четных и нечетных элементов

Осторожнее с фильтрами :even и :odd. Представим, что у нас есть следующий код:

Листинг 11.5. Неправильный код (фрагмент)

```
$(“tr:first”).css(“background-color”, “silver”);
$(“tr:last”).css(“background-color”, “red”);
$(“tr:even”).css(“background-color”, “green”);
$(“tr:odd”).css(“background-color”, “yellow”);
```

Мы хотим, чтобы заголовок таблицы (первый ряд) был окрашен в серый, последний (итоговый) ряд - в красный, четные ряды, чтобы были зелеными, а нечетные - желтыми. Результат выполнения этого кода изображен на рис. 11.5.



5	6	7	8
13	14	15	16

Рис. 11.5. Результат отличается от желаемого

Даже если вы изменили какие-то элементы, действие фильтров :odd и :even все равно будет на них распространяться, в итоге на рис. 11.5 мы получили только зеленый и желтый цвета, а красный и серый были переопределены фильтрами :odd и :even. Поэтому, чтобы получить именно то, что мы хотели, нужно сначала использовать фильтры :odd и :even, а потом уже :first и :last:

Листинг 11.6. Правильный код (фрагмент)

```
$(“tr:even”).css(“background-color”, “green”);  
$(“tr:odd”).css(“background-color”, “yellow”);  
$(“tr:first”).css(“background-color”, “silver”);  
$(“tr:last”).css(“background-color”, “red”);
```

Вот теперь результат тот, которого мы и добивались (рис. 11.6).

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Рис. 11.6. Правильное использование фильтров :odd и :even

11.3.4. Другие фильтры

Продолжим экспериментировать с нашей таблицей дальше. Фильтр :eq(индекс) может выбрать элемент с определенным индексом, например, чтобы выбрать третью ячейку таблицы, можно использовать следующий селектор:

```
$(“td:eq(2)”).css(“background-color”, “yellow”);
```

Нумерация элементов начинается с нуля, поэтому указан индекс 2.

Если нужно выбрать все элементы, начинающиеся с определенного индекса, например, все ячейки после третьей, тогда используйте фильтр :gt(индекс):

```
$(“td:gt(2)”).css(“background-color”, “yellow”);
```

Обратный фильтр - :lt(индекс). Он позволяет отыскать элементы с индексом, меньше, чем указанный:

```
$(“td:gt(3)”).css(“background-color”, “yellow”);
```

Также можно использовать фильтр `:not()`, который обычно используется для обращения условий других фильтров. Например, далее мы выберем все ячейки, кроме первой:

```
$( "td:not(:gt(3))" ).css ("background-color", "yellow");
```

Фильтр `:header` позволяет выбрать все элементы, являющиеся заголовками (теги `<h1>`, `<h2>` и т.д.):

```
$( "header" ).css ("border", "3px solid #f00");
```

Фильтровать элементы можно не только по их типу или позиции, но и по контенту. Представим, что вам нужно найти все элементы `<div>`, содержащие строку "Hello":

```
$( "div:contains('Hello')" ).css ("text-decoration", "bold");
```

Аналогично можно выбрать все абзацы, содержащие эту строку:

```
$( "p:contains('Hello')" ).css ("text-decoration", "bold");
```

Также полезен фильтр `:empty`, позволяющий найти пустые элементы - которые не содержат ни текста, ни дочерних элементов:

```
$( "td:empty" ).css ("background-color", "red");
```

Данный код находит все пустые ячейки и устанавливает для них красный цвет фона.

Фильтры `:visible` и `:hidden` позволяют найти, соответственно, видимые и скрытые элементы страницы. Следующий код выводит количество видимых и скрытых элементов в теге `<body>`:

```
alert("К-во видимых элементов ... " +
$( ":visible", document.body).length +
"\nК-во скрытых элементов ... " +
$( ":hidden", document.body).length);
```

Аналогично есть фильтры :enabled и :disabled. Первый позволяет отфильтровать включенные элементы, второй - отключенные (неактивные). Вот как можно установить значение для всех неактивных элементов ввода:

```
$(“input:disabled”).val(“DISABLED”);
```

При работе с формами вам также пригодятся фильтры :input и :password. Первый позволяет найти все поля ввода, второй - поля для ввода пароля (хотя пользы от него не очень много, поскольку обычно таких полей максимум два - и то в форме регистрации нового пользователя - пароль и его подтверждение). Примеры использования:

```
$(“:input”).css(“border”, “3px solid red”);
$(“:password”).css(“border”, “3px solid red”);
```

Фильтр :checked позволяет выбрать выбранные элементы **checkbox**. Следующий код позволяет вычислить количество включенных независимых переключателей:

```
<script>
$(function(){
    function counter() {
        var n = $("input:checked").length;
        $("div").text("Выбрано – " + n);
    }
    counter();
    $(":checkbox").click(counter);
});
</script>
```

Аналогично, фильтр :selected позволяет выбрать только выбранные опции списка. Далее с этими опциями можно что-то сделать, например, посчитать количество выбранных опций (если нужно, например, выбрать минимальное или максимальное количество), как в предыдущем примере.

11.4. Работа с атрибутами элементов

11.4.1. Методы attr() и removeAttr(): получение значения, установка и удаление атрибутов

Представим, что в нашем документе есть какой-нибудь элемент, пусть это будет ссылка. Вот как можно получить значение параметра этого аргумента:

```
var target = $("a").attr("target");
```

Методу attr() нужно передать имя атрибута, значение которого вы хотите получить.

В результате выполнения этого кода переменная **target** будет содержать значение атрибута **target** нашей ссылки. Конечно, забегая наперед, отмечу, что если в документе есть несколько ссылок, то, используя селекторы, описанные в предыдущей главе, вы можете уточнить, значение атрибута какой именно ссылки вы хотите получить.

Аналогичным образом вы можете изменить значение любого атрибута элемента страницы. Изменим адрес (атрибут **href**) ссылки:

```
$("a").attr("href", "http://nit.center");
```

Чтобы изменить значение атрибута, методу attr() нужно передать имя атрибута и новое значение. Как видите, метод attr() используется, как для получения, так и для установки значения атрибута.

В листинге 11.7 приведен полный пример кода получения и установки атрибута элемента:

Листинг 11.7. Пример получения и установки атрибута элемента

```
<html>
<head>
<title>Изменение атрибутов</title>
</head>

<body>
  <a href="http://mail.ru" target="_blank">Пример
изменения ссылки</a>
  <script src="https://code.jquery.com/jquery-
3.6.0.js"></script>
  <script>
    $(function() {
      let href = $("a").attr("href");
      alert("Текущий адрес: " + href + "\Новый адрес:
http://nit.center");
      $("a").attr("href", "http://nit.center");
    });
  </script>
</body>
</html>
```

Наша программа отобразит текущий и новый адрес ссылки, который будет установлен в результате ее выполнения. После выполнения программы наша ссылка будет указывать на сайт <http://nit.center>, а ранее она указывала на сайт <http://mail.ru>.

Обратите внимание, что метод attr() может также определить атрибуты, которые ранее отсутствовали в HTML-разметке. Например, посмотрим на нашу ссылку:

```
<a href="http://mail.ru" target="_blank">Пример
изменения ссылки</a>
```

В HTML-разметке были определены только два атрибута - `href` и `target`. Однако с помощью следующего оператора мы можем определить значение не описанного ранее атрибута, например, `title`:

```
$( "a" ).attr( "title", "Щелкни здесь" );
```

Если нужно определить несколько атрибутов, то можно использовать метод `attr()` так:

```
$( "a" ).attr( {  
    "title": "Щелкни здесь",  
    "href": "http://nit.center"  
} );
```

Для удаления атрибута, описанного или в HTML-разметке или с помощью метода `attr()` ранее, можно использовать метод `removeAttr()`:

```
$( "a" ).removeAttr( "title" );
```

11.4.2. Методы `addClass()`, `removeClass()`: работа со стилями

Современные страницы активно используют каскадные таблицы стилей (CSS), поэтому нельзя не рассмотреть методы для работы с классами:

- `addClass()` – добавляет класс;
- `removeClass()` – добавляет класс;
- `toggleClass()` – переключает класс;
- `hasClass()` – проверяет существование класса.

Давайте напишем программу, демонстрирующую работу с классами. Наша программа будет демонстрировать не только использование

вышеописанных методов, но также и установку обработчиков событий (в нашем случае - обработчиков нажатий кнопок).

Представим, что у нас есть таблица стилей. Она может быть определена или в отдельном CSS-файле или в секции `<style>` нашего документа. Для простоты кода будем считать, что она определена в нашем документе. Таблица небольшая и содержит всего две записи:

```
<style type="text/css">
div { border:2px solid #f00; }
.cl1 { background-color: green; color: white; }
</style>
```

Первый стиль - базовый для `div`, а второй стиль - стиль, с которым мы будем экспериментировать.

В HTML-код определены три тега `<div>` и четыре кнопки с идентификаторами `button1` - `button4`:

```
<div>Div 1</div>
<div>Div 2</div>
<div>Div 3</div>
<button id="button1">Добавить класс</button>
<button id="button2">Удалить класс</button>
<button id="button3">Переключить класс</button>
<button id="button4">Проверить класс</button>
```

Нажатия на ту или иную кнопку демонстрирует работу связанного метода. Назначить обработчик в JavaScript-коде для той или иной кнопки можно так:

```
$("#button1").click(function() {
    $("div").addClass("cl1");
});
```

Здесь мы с помощью метода click() задаем обработчик события, чтобы не определять его отдельно как функцию, мы используем безымянную функцию.

Теперь соберем все части программы воедино в листинге 11.8.

Листинг 11.8. Демонстрация работы со стилями

```
<html>
<head>
<title>Работа со стилями</title>
<style type="text/css">
div { border:2px solid red; }
.c11 { background-color: green; color: white; }
</style>
</head>
<body>

<div>Блок 1</div>
<div>Блок 2</div>
<div>Блок 3</div>
<button id="button1">Добавить класс</button>
<button id="button2">Удалить класс</button>
<button id="button3">Переключить класс</button>
<button id="button4">Проверить класс</button>
<script src="https://code.jquery.com/jquery-3.6.0.js"></script>
<script>
$(function() {
$("#button1").click(function() {
$("div").addClass("c11");
});

$("#button2").click(function() {
$("div").removeClass("c11");
});

$("#button3").click(function() {
$("div").toggleClass("c11");
});
});
```

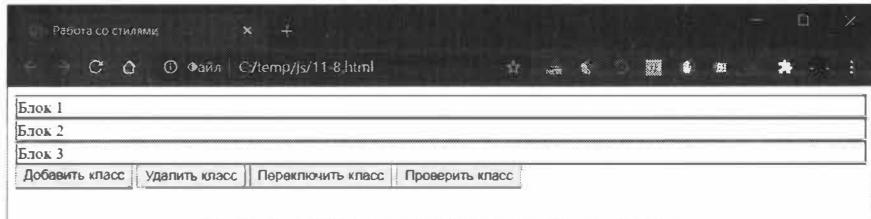


Рис. 11.7. Начальное состояние страницы

```
})
$( "#button4" ).click(function() {
alert($("div:last").hasClass("c11"));
});
});
</script>

</body>
</html>
```

На рис. 11.7 изображено начальное состояние нашей страницы - сразу после загрузки. Нажатие кнопки **Добавить класс** добавит класс к нашим `<div>` и страница будет выглядеть так, как показано на рис. 11.8.



Рис. 11.8. Класс применен (после нажатия Добавить класс)

Нажатие кнопку **Удалить класс** удаляет добавленный методом `addClass()` класс и возвращает нашу страницу к исходному состоянию (рис. 11.7). Кнопка **Переключить класс** позволяет переключаться между двумя классами, а кнопка **Проверить класс** проверяет, есть ли класс `c11` у последнего (`:last`) элемента `<div>`.

11.4.3. Методы `html()` и `text()`: работа с содержимым элементов

Получить содержимое элемента в виде HTML-кода или текста можно, соответственно, методами `html()` и `text()`. Рассмотрим небольшой пример (лист. 11.9):

Листинг 11.9. Пример использования метода `text()`

```
<html>
<head>
    <title>Метод text()</title>
</head>
<body>

<div id="hello">Привет <b>мир</b>!</div>
<script src="https://code.jquery.com/jquery-3.6.0.js"></
script>
<script>
$(function() {

    var txt = $("#hello").text();
    window.alert(txt);
});

</script>

</body>
</html>
```



Рис. 11.9. Использование метода `text()`

В HTML-коде есть <div> с идентификатором "hello". В коде программы мы получаем его содержимое как текст с помощью метода `text()`, а затем выводим с помощью диалогового окна. В результате окно отобразит следующий текст: "Привет мир!". А вот если бы мы использовали метод `html()`:

```
var txt = $("#hello").html();
```

то наше окно отобразило бы текст:

Привет мир!

Для установки содержимого элемента используются те же методы `text()` и `html()`. Рассмотрим следующий код:

```
$("#hello").text('<b>Привет!</b>');
```

В результате в `div` будет помещен следующий текст:

Привет!

А вот если мы будем использовать метод `html()`:

```
$("#hello").html('<b>Привет!</b>');
```

То текст будет таким:

Привет!

Если вам нужно установить содержимое элементов с HTML-разметкой, используйте метод `html()`. В большинстве случаев - это как раз то, что вам нужно. Если же нужно установить в качестве содержимого текст программы, HTML-код и подобные вещи, тогда используйте метод `text()`.

11.4.4. Метод `val()`: работа с атрибутом `value`

Метод `val()` позволяет получить значение атрибута `value`. Обычно этот метод принято использовать с полями ввода. В листинге 11.10

страница содержит поле ввода с именем `inp1` и кнопку, при нажатии которой диалоговое окно отображает значение поля ввода.

Листинг 11.10. Пример использования метода `val()`

```
<html>
<head>
<title>Метод val()</title>
</head>
<body>
  <input id="inp1" type="text" value="Ваше имя">
  <button id="button1">Получить значение!</button>
  <script src="https://code.jquery.com/jquery-
3.6.0.js"></script>
  <script>
    $(function() {
      $("#button1").click(function() {
        let value = $("#inp1").val();
        console.log(value);
      });
    });
  </script>
</body>
</html>
```

Рассмотрим еще один пример работы с атрибутом `value`. На этот раз мы с помощью jQuery узнаем, какой из радио-переключателей



Рис. 11.10. Результат использования метода `val()`

выбран. Чтобы получить значение выбранного переключателя нам нужно использовать вот такой код на основе фильтров:

```
var value = $("input:radio[name=os]:checked").val();
```

Здесь мы отфильтровываем поля ввода типа "radio" с именем "pet" и среди них выбираем активный переключатель (checked). Полный код приведен в листинге 11.11, а результат работы - на рис. 11.11.

Листинг 11.11. Использование метода val() для получения значения выбранного переключателя

```
<html>
<head>
<title>Выбранный переключатель</title>
</head>
<body>
<p>Есть ли у вас животное:</p>
<input type="radio" name="pet" value="cat"
checked="checked" />Кошка
<input type="radio" name="pet" value="dog" />Собака
<input type="radio" name="pet" value="other" />Другое
<button id="button1">Результат!</button>
<script src="https://code.jquery.com/jquery-
3.6.0.js"></script>
<script>
$(function(){
    $("#button1").click(function(){
        var value = $("input:radio[name=pet]:checked").val();
        window.alert(value);
    });
});
</script>

</body>
</html>
```



Рис. 11.11. Отображение выбранного животного

11.5. Визуальные эффекты

11.5.1. Методы `hide()`, `show()`, `toggle()`: управление видимостью

Методы `hide()`, `show()` и `toggle()` используются для управления видимостью элементов страницы, а именно: метод `hide()` позволяет скрыть элемент, `show()` - показать элемент, а `toggle()` - переключает состояние видимости/невидимости элемента. Пример использования этих методов приведен в листинге 11.12.

Листинг 11.12. Методы `hide()`, `show()` и `toggle()`

```
<html>
<head>
<title>Методы hide(), show() и toggle() </title>
</head>
<body>

<button id="show">Показать</button>
<button id="hide">Скрыть</button>
<button id="toggle">Переключить</button>
<p>
<div>Lorem ipsum dolor sit amet, consectetur adipiscing
elit. Fusce ut justo consequat, sollicitudin tortor
eu, ultrices risus. Nullam nec elit cursus metus rutrum
laoreet. Maecenas blandit arcu eget enim tincidunt
pellentesque. Phasellus turpis libero, egestas nec diam
vel, dapibus aliquam odio.</div>
```

```
<script src="https://code.jquery.com/jquery-3.6.0.js"></script>
<script>
$(function() {
    $("#show").click(function() {
        $("div").show();
    });

    $("#hide").click(function() {
        $("div").hide();
    });

    $("#toggle").click(function() {
        $("div").toggle();
    });
});
</script>

</body>
</html>
```

Итак, у нас есть три кнопки, которые управляют видимостью абзаца текста, заключенного в тег `<div>`. Нажатие кнопки **Скрыть** вызывает метод `hide()`, который скрывает заключенный в `<div>` текст, нажатие кнопки **Показать** вызывает одноименный метод и снова отображает текст. Нажатие кнопки **Переключить** переключает видимость/невидимость блока текста.

11.5.2. Методы `slideUp()`, `slideDown()` и `slideToggle()`: скольжение

Методы `slideUp()`, `slideDown()` и `slideToggle()` используются для организации эффекта скольжения. Эти методы также можно использовать для отображения/скрытия элемента страницы, но они выполняют скрытие/отображение плавно. Пример, приведенный в листинге 11.13, демонстрирует работу этих методов.

Листинг 11.13. Плавное сокрытие/отображение элемента страницы

```
<html>
<head>
<title>Скользжение</title>
<style type="text/css">
    div { border:1px dotted #369; padding:5px; color: blue;
}
</style>
</head>
<body>

<button id="hide">Скрыть</button>
<button id="show">Показать</button>
<button id="toggle">Переключить</button>

<p>
<div id="text">
<p>Lorem ipsum dolor sit amet, consectetur adipiscing
elit. Fusce ut justo consequat, sollicitudin tortor
eu, ultrices risus. Nullam nec elit cursus metus rutrum
laoreet. Maecenas blandit arcu eget enim tincidunt
pellentesque. Phasellus turpis libero, egestas nec diam
vel, dapibus aliquam odio. Curabitur molestie mauris in
arcu tincidunt ornare.</p>
</div>
<script src="https://code.jquery.com/jquery-3.6.0.js"></
script>
<script>
$(function() {

    $("#hide").click(function(){
        $("#text").slideUp(1500);
    });

    $("#show").click(function(){
        $("#text").slideDown(1500);
    });
})
```

```
$( "#toggle" ).click(function() {
    $( "#text" ).slideToggle(1500);
});  
});  
</script>  
  
</body>  
</html>
```

Все работает, как и в предыдущем примере, но для большей наглядности был добавлен стиль для `<div>` и кнопки приведены в несколько другом порядке. Нужно отметить, что методы `slideUp()`, `slideDown()` и `slideToggle()`, как и методы `hide()`, `show()` и `toggle()`, могут принимать дополнительные параметры: функцию, которая будет выполнена при выполнении соответствующего метода и название эффекта плагина `easing`. Данный параметр можно указать только, если вы подключили плагин `easing` (<https://jqueryui.com/easing/>). Например:

```
slideUp(2000, 'linear', f1);
```

Здесь 2000 - как обычно, задержка в миллисекундах, 'linear' - название эффекта (поддерживаются только эффекты `swing` и `linear`), а f1 - название функции, которая будет вызвана.

11.5.3. Методы `fadeOut()`, `fadeIn()` и `fadeToggle()`: эффект затухания

Методы `fadeOut()`, `fadeIn()` и `fadeToggle()` также позволяют отображать/скрывать элемент, но с эффектом затухания. Довольно интересный эффект и я рекомендую испытать его на практике. Используются эти методы аналогично предыдущим, поэтому за основу взят листинг 11.13, но JavaScript-код будет следующим (листинг 11.14):

Листинг 11.14. Демонстрация эффекта затухания

```
<script>
$(function() {
    $("#hide").click(function() {
        $("#text").fadeOut(1500);
    });

    $("#show").click(function() {
        $("#text").fadeIn(1500);
    });

    $("#toggle").click(function() {
        $("#text").fadeToggle(1500);
    });
});
</script>
```

Методы `fadeOut()`, `fadeIn()` и `fadeToggle()` также, кроме задержки затухания, принимают эффект плагина **easing** и имя функции, которая будет выполнена. Конечно, эти два параметра необязательны и все работает прекрасно и без них.

11.5.4. Метод `fadeTo()`: плавное изменение прозрачности

Метод `fadeTo()` используется для плавного изменения прозрачности элемента (до заданного значения или вплоть до его исчезновения). Как говорится, лучше один раз увидеть, поэтому попробуйте выполнить пример из листинга на практике.

Листинг 11.15. Плавное изменение прозрачности

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=utf-8" />
```

```
<title>Изменение прозрачности элемента</title>
<style type="text/css">
div { border:1px dotted #379; padding:5px; color: blue; }
</style>
</head>
<body>
<button>Нажми меня</button>
<p>Lorem ipsum dolor sit amet, consectetur adipiscing
elit. Fusce ut justo consequat, sollicitudin tortor
eu, ultrices risus. Nullam nec elit cursus metus rutrum
laoreet. Maecenas blandit arcu eget enim tincidunt
pellentesque. Phasellus turpis libero, egestas nec diam
vel, dapibus aliquam odio. Curabitur molestie mauris in
arcu tincidunt ornare.</p>
<script src="https://code.jquery.com/jquery-3.6.0.js"></
script>
<script>
$(function(){
  $("button").one("click", function(){
    $("p:first").fadeTo("slow", 0.30, function() { console.
log("Готово!"); });
  });
});
</script>

</body>
</html>
```

Сначала с помощью метода `one()` мы устанавливаем так называемый одноразовый обработчик события - он будет вызван только один раз - когда вы впервые нажмете на кнопку. Это связано с тем, что повторно изменять прозрачность опять до фиксированного значения 0.30 нет смысла, поскольку этот уровень прозрачности уже будет задан при первом нажатии на кнопку, и вы просто не увидите разницы.

При нажатии на кнопку будет вызван метод `fadeTo()` для абзаца. Первый параметр - это скорость изменения прозрачности и он может принимать значения: `slow` (медленно), `normal` (обычно) и `fast` (бы-

стро). Как по мне, наиболее удачный вариант - значение slow. Второй параметр - это степень прозрачности, может принимать значения от 0 (полностью прозрачный) до 1 (полностью непрозрачный). Третий параметр - функция, которая будет вызвана, как только прозрачность элемента будет доведена до указанного значения. Особого толку в этой функции нет, и здесь она отображает сообщение "Готово!" сугубо в демонстрационных целях. Кстати, аналогичным образом вы можете указывать функции и в предыдущих методах.

11.5.5. Метод animate(): простейшая анимация

Метод `animate()` можно использовать для создания простейшей анимации. В нашем случае мы будем передвигать по странице величайшее произведение искусства - черный квадрат. Представим, что у нас есть черный квадрат с некоторыми начальными координатами и нам нужно с помощью кнопок Left/Right/Up/Down перемещать этот квадрат по всему документу. Как это сделать? Конечно, достаточно легко установить свойства `left` и `top` элемента для изменения его позиции: черный квадрат сразу же перейдет на заданную позицию. Но это не интересно. Интересно сделать перемещение квадрата плавным. А для этого как раз пригодится метод `animate()`.



Рис. 11.12. Перемещение черного квадрата

Первый параметр этого метода принимает список, состоящий из пар ключ/значения различных CSS-свойств. Мы будем изменять только свойства **left** и **top**. Второй параметр определяет скорость анимации, опять мы будем использовать значение *slow*.

Нажатие кнопок **Влево/вправо** будет изменять на 30 пикселей значение свойства **left**, что обеспечивает горизонтальное перемещение квадрата. А нажатие кнопок **Вверх/Вниз** изменяет свойство **top** для перемещения по вертикали (рис. 11.12).

Полный код страницы представлен в листинге 11.16.

Листинг 11.16. Перемещение квадрата

```
<html>
<head>
<title>Перемещение квадрата</title>
<meta http-equiv="Content-Type" content="text/html;
charset=utf-8" />
<style type="text/css">
div {
    position:absolute;
    background-color:#000;
    left:70px;
    top:70px;
    width:200px;
    height:200px;
    margin:10px;
}
</style>
</head>
<body>
    <button id="left">Влево</button>
    <button id="right">Вправо</button>
    <button id="up">Вверх</button><button id="down">Вниз</
button>
    <div class="block"></div>
```

```
<script src="https://code.jquery.com/jquery-
3.6.0.js"></script>
<script>
$(function() {

    $("#right").click(function() {
        $(".block").animate({"left": "+=50px"}, "slow");
    });
    $("#left").click(function() {
        $(".block").animate({"left": "-=50px"}, "slow");
    });
    $("#down").click(function() {
        $(".block").animate({"top": "+=50px"}, "slow");
    });
    $("#up").click(function() {
        $(".block").animate({"top": "-=50px"}, "slow");
    });

});
</script>

</body>
</html>
```

11.6. Работа с CSS

11.6.1. Метод `css()`: получение и установка значения CSS-свойства

Метод `css()` позволяет, как получать, так и устанавливать значения CSS-свойства. Чтобы получить значение свойства, нужно указать его имя в качестве первого параметра метода `css()`. Для установки значения свойства нужно установить не только название свойства, но и его новое значение. Пример:

```
<style type="text/css">
div { width:150px; height:150px; margin:10px; float:left;
}
</style>
...
$(function() {
    let old_width = $("div").css("width");
    window.alert(old_width);

    let new_width = "110px";
    $("div").css("width", new_width);
});
```

Сначала мы получаем старое значение ширины (свойство width) в переменную old_width. Затем выводим значение переменной old_width. Пока вы не нажмете кнопку **OK** в диалоговом окне, вы будете видеть, как выглядит блок текста до (рис. 11.13, слева) и после (рис. 11.13, справа) изменения его свойств.



Рис. 11.13. Демонстрация использования метода css()

Полный код страницы приведен в листинге 11.17.

Листинг 11.17. Изменение значений CSS-свойств

```
<html>
<head>
<title>Пример изменения CSS-свойства</title>
<meta http-equiv="Content-Type" content="text/html;
charset=utf-8" />
<style type="text/css">
div {
    width:25px;
    height:25px;
```

```
margin:2px;
float:left;
border: 1px solid #379; }
</style>
</head>
<body>
    <div>Привет всем!</div>
    <script src="https://code.jquery.com/jquery-
3.6.0.js"></script>
    <script>
        $(function() {

            let old_width = $("div").css("width");
            window.alert(old_width);

            let new_width = "110px";
            $("div").css("width", new_width);

        });
    </script>

</body>
</html>
```

Рассмотрим еще один пример использования метода **css**. Представим, что нам нужно изменить фон какого-нибудь элемента страницы. JavaScript-код будет таким:

```
<script>
$(function(){
    $("button").click(function () {
        let color = $("input").val();
        $("div").css("background-color", color);
    });
});
```

При нажатии кнопки мы получаем значение цвета из поля `input`. Полученное значение мы устанавливаем в качестве цвета фона `<div>`. HTML-код выглядит так:

```
<input type="text" />
<button>Применить</button>
<div></div>
<div></div>
```

Чтобы пример работал, как задумывалось нужно еще подключить таблицу стилей:

```
<style type="text/css">
div {
    width:250px;
    height:150px;
    margin:10px;
    float:left;
    border:1px solid #379;
}
</style>
```

Данная таблица устанавливает размеры области `<div>`, чтобы изменение фона было заметно.

11.6.2. Другие методы

Большинство задач, связанных с изменением CSS, можно выполнить и через метод `css()`. Но я не мог не рассказать о других методах, которые есть в jQuery. Сейчас мы напишем приложение, демонстрирующее многие из этих методов. Заметьте, что все эти методы можно использовать, как для получения, так и для установки значений. Например, метод `width()`, если вызван без параметров, возвращает ширину элемента. Если же передать ему параметр, то он установит ширину.

Следующий пример довольно прост, и вы сможете разобраться с ним без каких-либо комментариев. Заодно он послужит наглядным примером изменения размеров и других CSS-свойств элемента `<div>` (лист. 11.18).

Листинг 11.18. Демонстрация методов для работы с CSS

```
<html>
<head>
<title>Демонстрация методов для работы с CSS</title>
<meta http-equiv="Content-Type" content="text/html;
charset=utf-8" />
<style type="text/css">
div {
    width:300px; height:150px;
    margin:15px; padding:10px;
    border:1px dotted #000;
    overflow:hidden;
}
button { width:180px; }
</style>
</head>
<body>
<button>Получить width</button>
<button>Получить height</button>
<button>Получить innerWidth</button>
<button>Получить innerHeight</button>
<button>Получить outerWidth</button>
<button>Получить outerHeight</button>
<br /><br />
<input type="text" />
<button>Установить width</button>
<button>Установить height</button>
<div>Lorem ipsum dolor sit amet, consectetur adipiscing
elit. Vestibulum pharetra libero id neque ultricies
facilisis. Curabitur diam justo, molestie ut tellus non,
eleifend hendrerit massa.</div>
<script src="https://code.jquery.com/jquery-3.6.0.js"></
script>
```

```
<script>
$(function(){
    $("button:eq(0)").click(function(){
        $("input").val($(".div").width());
    });
    $("button:eq(1)").click(function(){
        $("input").val($(".div").height());
    });
    $("button:eq(2)").click(function(){
        $("input").val($(".div").innerWidth());
    });
    $("button:eq(3)").click(function(){
        $("input").val($(".div").innerHeight());
    });
    $("button:eq(4)").click(function(){
        $("input").val($(".div").outerWidth(true));
    });
    $("button:eq(5)").click(function(){
        $("input").val($(".div").outerHeight());
    });
    $("button:eq(6)").click(function(){
        $(".div").width($(".input").val()*1);
    });
});
```

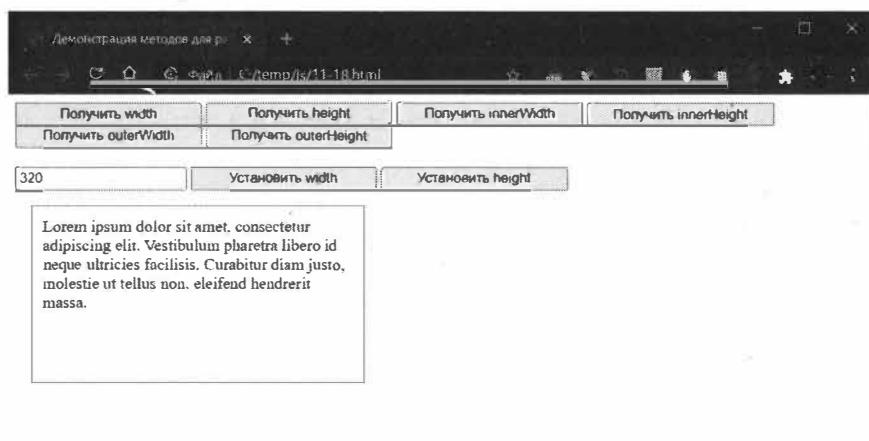


Рис. 11.14. Демонстрация методов jQuery

```
$("button:eq(7)").click(function() {
    $("div").height($(".input").val()*1);
});
});
</script>

</body>
</html>
```

Примечание. Значения, полученные из элемента `input`, мы умножаем на единицу. Так мы преобразуем строковое значение, возвращаемое методом `val()`, в число, которое принимают методы `width(val)` и `height(val)` принимают число.

Примечание. Ознакомиться с остальными методами можно по адресу <https://api.jquery.com/category/css/>.

11.7. Работа с содержимым элемента

11.7.1. Операции над содержимым элементов

Некоторые операции над содержимым элементов, в частности, методы `text()` и `html()`, которые были описаны ранее. Однако, это не единственныe методы, доступные в jQuery. Методы `text()` и `html()` позволяют, как получить, так и установить значение элементов. Когда эти методы устанавливают значение элемента, то предыдущее значение полностью перезаписывается. Это не всегда нужно. Иногда нужно добавить в элемент новое содержимое, сохранив при этом старое. Можно, конечно, получить сначала старое значение - с помощью метода `text()` или `html()`, добавить к этому значению дополнительное, а потом с помощью этих же методов установить новое значение.

Но это - лишняя работа. Гораздо проще использовать метод append().
Вот как можно использовать этот метод:

```
$( "p" ).append( "<b>Пока!</b>" );
```

Если же, наоборот, нужно добавить новое содержимое перед старым содержимым, тогда используется метод prepend():

```
$( "p" ).prepend( "<b>Hello! </b>" );
```

11.7.2. Методы appendTo() и prependTo()

Представим, что у нас есть какие-то элементы и нужно поместить их в какой-то определенный элемент, причем они должны быть вставлены после или до существующего содержимого этого элемента. Данную задачу можно решить с помощью методов appendTo() и prependTo().

Давайте, посмотрим, как работают эти методы на примере метода appendTo(). Метод prependTo() работает аналогично, но добавляет элементы до существующего контента, а не после него. Рабочий пример представлен в листинге 11.19.

Листинг 11.19. Демонстрация работы метода appendTo()

```
<html>
<head>
<title>Метод appendTo()</title>
</head>
<body>
<div id="bye">Пока!</div>
<button>Вставить</button>
<div id="msg">Привет! </div>
<script src="https://code.jquery.com/jquery-3.6.0.js"></script>
<script>
$(function(){
    $("button").click(function() {
        $("#bye").appendTo("#msg");
    });
});
```

```
});  
</script>  
  
</body>  
</html>
```

При нажатии кнопки **Вставить** блок с id = "bye" будет присоединен к <div> с идентификатором "msg", следовательно, надпись "Пока!" появится под надписью "Привет!" (см. рис. 11.15).

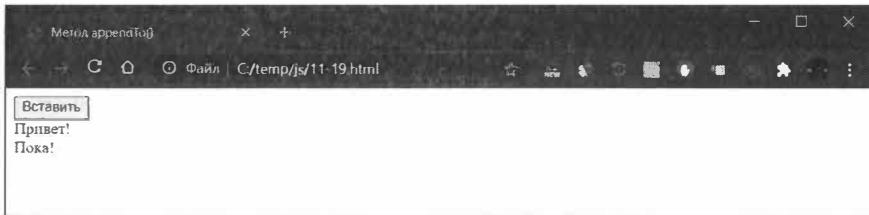


Рис. 11.15. Результат работы сценария из листинга 11.19

11.7.3. Методы after() и before()

Методы after() и before() используются, когда нужно вставить некоторый HTML-код после и до определенного элемента.

```
$( "p" ).before( "<b>до текста</b>" );  
$( "p" ).after( "<b>после текста</b>" );
```

Пример использования этих методов приведен в листинге 11.20.

Листинг 11.20. Демонстрация использования методов after() и before()

```
<html>  
<head>  
<title>Методы after() и before()</title>  
</head>  
<body>  
  
<p>Текст
```

```
<script src="https://code.jquery.com/jquery-3.6.0.js"></script>
<script>
$(function() {
    $("p").before("<b>до текста</b>");
    $("p").after("<b>после текста</b>");
});
</script>

</body>
</html>
```

Результат изображен на рис. 11.16.

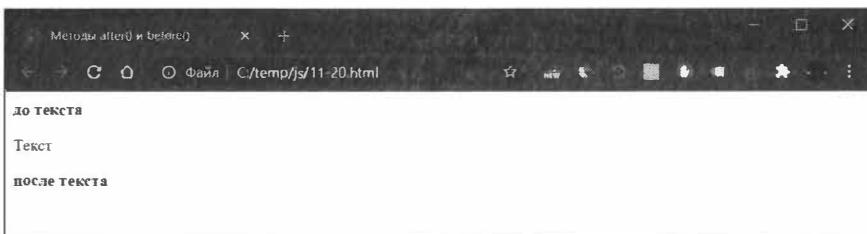


Рис. 11.16. Демонстрация использования методов `after()` и `before()`

11.7.4. Методы `wrapInner()` и `wrapAll()`

Методы `wrapInner()` и `wrapAll()` используются для изменения разметки HTML-страницы. Представим, что у нас есть несколько элементов `<a>`, которые следуют друг за другом. Данные элементы можно поместить в нумерованный (или ненумерованный) список, поместив каждый `<a>` в тег ``. Содержимое в тегах `<a>` (текст ссылки) нужно выделить курсивом (заключить в тег `<i>`). Задача, если ее решать средствами JavaScript, довольно сложная, поскольку требует или сложных регулярных выражений или громоздкого JavaScript-кода без регулярных выражений.

К счастью, у нас есть jQuery, которая существенно все упрощает. В листинге 11.21 показано, как элегантно jQuery справляется с поставленной задачей.

Листинг 11.21. Изменение HTML-разметки

```
<html>
<head>
<title>Изменение разметки</title>
</head>
<body>
  <a href="http://www.discovery.com/">Discovery</a>
  <a href="http://nit.center/">Nit.Center</a>
  <a href="http://www.booking.com/">Booking.com</a>
  <a href="http://www.tripadvisor.com/">TripAdvisor</a>
  <a href="http://www.livejournal.com/">LiveJournal</a>
  <button>Нажми меня</button>
  <script src="https://code.jquery.com/jquery-
3.6.0.js"></script>
  <script>
    $(function() {
      $("button").click(function() {
        $("a").wrapInner("<i></i>").wrap("<li></li>");
        $("li").wrapAll("<ol></ol>");
      });
    });
  </script>

</body>
</html>
```

Наша сложнейшая задача решается методами `wrapInner()`, `wrapAll()` и `wrap()`. Метод `wrapInner()` "берет" внутренний контент тегов `<a>`, то есть текст ссылки, и обрамляет его тегами `<i>` и `</i>`. Именно этот метод делает внутренний текст ссылки курсивным. Далее методом `wrap()` мы обрамляем каждую ссылку в теги ``. Наконец, методом `wrapAll()`, который применяется не к одному по отдельности, а ко всем сразу тегам ``, помещаем все элементы списка в нумерованный список. Результат работы этого сценария - на рис. 11.17.

Отметить действие метода `wrap()` может метод `unwrap()`. В нашем случае данный метод преобразует список ссылок просто в строку,



Рис. 11.17. Изменение HTML-разметки

котором ссылки будут следовать друг за другом (без пробелов или других разделителей). Пример использования этого метода приведен в листинге 11.22.

Листинг 11.22. Метод Unwrap

```
<html>
<head>
<title>Метод unwrap</title>
</head>
<body>
    <a href="http://www.discovery.com/">Discovery</a>
    <a href="http://www.nationalgeographic.com/">National
Geographic.com</a>
    <a href="http://www.booking.com/">Booking.com</a>
    <a href="http://www.tripadvisor.com/">TripAdvisor</a>
    <a href="http://www.livejournal.com/">LiveJournal</a>

    <button id="b1">Метод Wrap</button>
    <button id="b2">Метод Unwrap</button>
    <script src="https://code.jquery.com/jquery-
3.6.0.js"></script>
    <script>
        $(function() {
            $("#b1").click(function() {
                $("a").wrapInner("<i></i>").wrap("<li></li>");
                $("li").wrapAll("<ol></ol>");
            });
        });
    </script>
```

```
$( "#b2" ).click(function() {
    $("a").unwrap();
});
});
</script>

</body>
</html>
```

У нас есть две кнопки. Нажатие первой кнопки строит нумерованный список, изображенный на рис. 11.17. Нажатие второй кнопки отменяет создание нумерованного списка.

11.7.5. Поиск и замена элементов

Представим еще одну практическую ситуацию. Пусть у нас есть страница, где нужно заменить теги `<p>` тегами `<div>`. Опять-таки для этой задачи очень эффективно использовать jQuery. Рассмотрим следующий очень компактный код:

```
$(function() {
    $("p").replaceWith("<div>" + $("p").text() + "</div>");
});
```

Данный код заменяет каждый тег `<p>` тегом `<div>`. Также можно использовать и метод `replaceAll()`:

```
$(<div>).replaceAll("p");
```

Чтобы лучше понять, как работает метод `replaceWith()` рассмотрим листинг 11.23.

Листинг 11.23. Демонстрация работы метода `replaceWith()`

```
<html>
<head>
<title>Использование replaceWith()</title>
<style type="text/css">
p {
border:1px solid green;
```

```
color:#169;
}
div {
border:2px solid red;
color:#02f;
}
</style>

</head>
<body>
<p>Первый параграф</p>
<p>Второй параграф</p>
<div>Div</div>
<script src="https://code.jquery.com/jquery-3.6.0.js"></
script>
<script >
$(function() {
    $("p").click(function() {
        $(this).replaceWith("<div>" + $(this).text() + "</
div>");
    });
});
</script>

</body>
</html>
```

Чтобы было визуально заметно, что параграф заменен на `<div>`, для тегов `<p>` и `<div>` в таблице стилей задано разное оформление. При щелчке на параграфе создается событие `click`, обработчик которого преобразует параграф `<p>` в `<div>`. Иллюстрации не будет, поскольку этот пример нужно попробовать выполнить "вживую", на листах черно-белой книги вы не поймете, что произошло на самом деле.

11.7.6. Другие полезные методы

Кроме описанных ранее методов наверняка вам пригодятся следующие полезные методы:

- `empty()` – очищает элемент, удаляя все его содержимое. Сам элемент при этом остается на странице
- `remove()` – удаляет элемент
- `clone()` – клонирует элемент. Клонируются не только атрибуты (свойства) элемента, но также и связанные с ним обработчики событий

На этом все, а в следующей главе мы поговорим о перемещении по DOM-элементам.

11.8. Обработка событий

Ранее было показано, как установить обработчик кнопки с помощью метода `onClick()`, но в jQuery имеются и другие методы обработки событий (см. табл. 11.1).

Таблица 11.1. Методы обработки событий jQuery

Метод	Описание
<code>blur()</code>	Позволяет установить обработчик события "blur.." Если этот метод вызван без параметров, то он порождает событие "blur".
<code>"change()</code>	Устанавливает обработчик события <code>change</code> или порождает это событие, если вызван без параметров
<code>click()</code>	Устанавливает обработчик события <code>click</code> или порождает это событие, если вызван без параметров

contextmenu	<p>Устанавливает обработчик события contextmenu или порождает это событие, если вызван без параметров. Пример кода:</p> <pre data-bbox="405 292 667 374"><div id="target"> Правый клик </div></pre> <p>JavaScript-код:</p> <pre data-bbox="405 454 969 669">\$("#target").contextmenu(function() { alert("Обработчик для .contextmenu() вызван.. "); }); ... \$("#target").contextmenu();</pre>
click()	<p>Устанавливает обработчик события dblclick или порождает это событие, если вызван без параметров</p>
error()	<p>Устанавливает обработчик события errort или порождает это событие, если вызван без параметров</p>
focus(), focusin(), focusout()	<p>Позволяют установить обработчик соответствующих событий или вызвать их</p>
keypress(), keyup(), keydown(), keyup()	<p>Позволяют установить обработчик соответствующих событий клавиатуры или вызвать их</p>
mouseenter(), mouseleave(), mousemove(), mouseout(), mouseover(), mouseup()	<p>Позволяют установить обработчик соответствующих событий мыши или вызвать их</p>

Также можно использовать универсальный метод `on()`, который используется для установки обработчика одного или нескольких событий, прототип метода выглядит так:

```
on( events [, selector ] [, data ], handler )
```

Пример использования метода:

```
function notify() {  
    alert( "Привет!" );  
}  
$( "button" ).on( "click", notify );
```

11.9. AJAX и jQuery

AJAX (*Asynchronous Javascript and XML*) – технология передачи данных без перезагрузки страницы. Представим себе витрину обычного Интернет-магазина. Вы нажимаете купить и выбранный товар должен попасть в корзину. Ранее код товара передавался на сервер, далее браузер получал ответ от сервера – или опять витрину, где изменена надпись – количество элементов в корзине – или же саму корзину со списком добавленных в нее товаров (зависит от настроек магазина). Смысл в том, что происходила перезагрузка страницы. Посредством AJAX можно передать информацию на сервер без перезагрузки страницы. Большая часть страницы останется без изменения, будет изменено только количество товаров в корзине. Это значительно экономит ресурсы, как пользователя (Интернет-трафик, который все еще не всегда бесплатный), так и сервера (значительно снижается нагрузка, поскольку вместо страницы, размер которой в среднем составляет от 3-4 Мб нужно передать всего лишь несколько байтов информации – количество элементов в корзине и служебные заголовки).

Технология AJAX может работать и на чистом JavaScript, без подключения jQuery или любой другой сторонней библиотеки. Однако

именно jQuery позволяет сделать работу с AJAX удобнее. Далее мы рассмотрим методы jQuery для взаимодействия с AJAX.

11.9.1. Загрузка страницы. Метод `load()`

Рассмотрение методов, связанных с AJAX, мы начнем с метода `load()`, который позволяет загрузить веб-страницу целиком. Синтаксис этого метода следующий:

```
load(url, [data], [complete(responseText, textStatus, XMLHttpRequest)])
```

Обязательный лишь первый параметр - URL загружаемой страницы. Сейчас мы напишем простейшее приложение, загружающее сторонней HTML-код страницы и выводящее его в какой-то `div` на текущей странице. Код приложения прост и приведен в листинге 11.24.

Листинг 11.24. Пример использования метода `load()`

```
<html>
<head>
<title>Использование Load()</title>
<meta http-equiv="Content-Type" content="text/html;
charset=utf-8" />
<style type="text/css">
div {
    width:700px; height:600px;
    padding:10px; margin:10px;
    border:3px double #379;
    overflow:auto;
}
</style>
</head>
<body>
<div id="target"></div>
<button>Загрузить</button>
<button>Очистить</button>
```

```
<script src="https://code.jquery.com/jquery-3.6.0.js"></script>
<script>
$(function () {
    $("button:first").click(function(){
        $("#target").load("http://nit.center/test.htm");
    });
    $("button:last").click(function(){
        $("#target").empty();
    });
});
</script>
</body>
</html>
```

Посмотрим, что делает приложение. При нажатии кнопки **Загрузить** метод `load()` получает код страницы `test.htm` и выводит его в `div` с **id target**. Все как бы просто. Но при всей своей простоте у приложения есть нюансы. Оно будет нормально работать в Firefox (по крайней мере, до версии 53 включительно), но в Chrome есть нюансы.

Если страница, указанная в `load()`, находится на другом домене или же будет просто находиться на локальном компьютере, то загрузить стороннюю страницу не получится:

- Если вы откроете файл примера на локальном компьютере, то увидите сообщение, что нужно использовать следующие схемы протокола: `http`, `https` и некоторые другие.
- Если файл `11-23.html` поместить на сервер и попытаться открыть страницу, которая находится на другом домене, то сообщение об ошибке будет таким `No 'Access-Control-Allow-Origin' header is present on the requested resource.`

С первой ошибкой все ясно, а вот что делать со второй? Политика безопасности браузеров запрещает выполнение кроссдоменных AJAX запросов. Чтобы это обойти, как было замечено выше, нужно добавить заголовок `Access-Control-Allow-Origin`, но возвращать

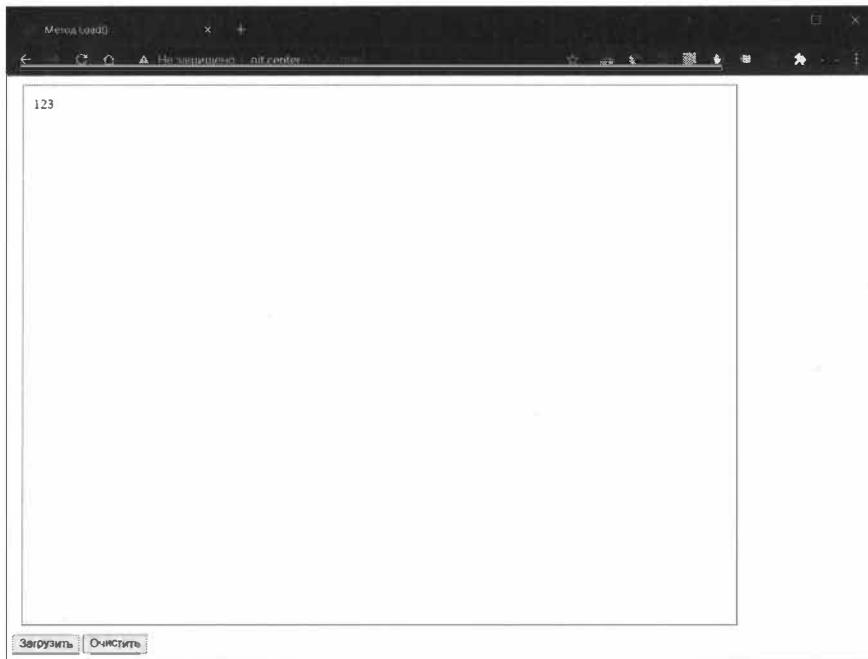


Рис. 11.18. Выполнение сценария (Chrome, версия 88.0, март 2021 года)

его должен домен, к которому вы обращаетесь. С обычной HTML-страничкой такой трюк не получится, но если вы хотите загрузить страницу, которая генерируется PHP-сценарием, то в начало сценария нужно добавить оператор:

```
header("Access-Control-Allow-Origin: *");
```

Проблема в том, что если у вас нет доступа к сценарию, который выводит эту страничку, то и загрузить ничего не получится. То есть метод `load()` будет полезен при загрузке только своих страниц, а не чужих.

Метод `load()` можно использовать не только для загрузки всей страницы, но и определенной ее части. Представим, что в загружаемой странице есть несколько абзацев:

<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit

Для загрузки последнего абзаца можно использовать следующий код:

```
<script>
$(function () {
    $("button:first").click(function(){
        $("#target").load("nit.html p:last", function(){
            alert("Готово!");
        });
    });
    $("button:last").click(function(){
        $("#target").empty();
    });
});
</script>
```

Мы использовали `p:last`, чтобы указать, что нам нужен последний из приведенных абзацев (`<p>`).

11.9.2. Получение данных от сервера с использованием запроса POST

Довольно часто на практике возникает задача получения каких-то данных от сервера, используя POST-запрос. Для этого в jQuery предусмотрен метод `$.post`. Его прототип выглядит так:

```
$.post(url, [data], [success(data, textStatus, jqXHR)], [dataType])
```

Представим, что у нас есть сценарий PHP, принимающий значения `login` и `password` и возвращающий какую-то информацию. В нашем случае он просто будет возвращать переданные ему значения:

```
<?php
header('Content-Type: text/html; charset=utf-8');
```

```
if($_SERVER['HTTP_X_REQUESTED_WITH'] == 'XMLHttpRequest')
{
    if($_POST) {
        echo 'Login: ' . $_POST['login'] . ', password: ' .
        $_POST['password'];
    }
}
?>
```

Передать POST-запрос можно так:

```
$.post("post.php",
    { login: "mark", password: "123456" },
    function(data) {
        $("div").text(data);
    });
});
```

Здесь post.php - имя сценария, которому передаются значения *login* и *password*. Также мы устанавливаем функцию-обработчик ответа. В нашем случае она, получив ответ, выведет его в <div> на странице. Полный код приложения приведен в листинге 11.25.

Примечание. Запускать наше приложение нужно на веб-сервере с поддержкой PHP. Если просто открыть 11-24.html в браузере с локального компьютера, приложение работать не будет (или нужно post.php разместить на удаленном сервере и указать полный URL к нему).

Листинг 11.25. Отправка POST-запроса и получение ответа сервера

```
<html xmlns="http://www.w3.org/1999/xhtml" lang="ru"
xml:lang="ru">
<head>
<title>POST-запрос</title>
<meta http-equiv="Content-Type" content="text/html;
charset=utf-8" />
<style type="text/css">
div {
    width:300px; height:250px;
```

```
padding:15px; margin:15px;
border:3px double blue;
overflow:auto;
}
</style>
</head>
<body>
<div></div>
<button>Отправить POST-запрос</button>
<button>Очистить</button>
<script src="https://code.jquery.com/jquery-3.6.0.js"></script>
<script>
$(function () {
    $("button:first").click(function() {
        $.post("post.php",
            { login: "mark", password: "123456" },
            function(data) {
                $("div").text(data);
            });
    });
    $("button:last").click(function() {
        $("div").empty();
    });
});
</script>

</body>
</html>
```

11.9.3. Получение данных от сервера с использованием запроса GET

Попробуем решить аналогичную задачу, но используя GET-запрос. Для этого используется метод `$.get`:

```
$.get(url, [data], [success(data, textStatus, jqXHR)], [dataType])
```

Код JavaScript будет таким же:

```
<script>
$(function () {
    $("button:first").click(function(){
        $.get("get-data.php",
            { login: "mark", password: "123456" },
            function(data){
                $("div").text(data);
            });
    });
    $("button:last").click(function(){
        $("div").empty();
    });
});
</script>
```

Соответственно, сценарий post.php нужно превратить в get-data.php так:

```
<?php
header('Content-Type: text/html; charset=utf-8');
if($_SERVER['HTTP_X_REQUESTED_WITH'] == 'XMLHttpRequest')
{
    if($_GET) {
        echo 'Login: ' . $_GET['login'] . ', password: ' .
        $_GET['password'];
    }
}
?>
```

Все точно также, только он получает данные, используя GET, а не POST.

11.9.4. Загрузка и выполнение JS-файлов

Метод `$.getScript()` используется для получения и выполнения JavaScript-файлов. При этом JavaScript-файлы могут быть, как локальными, так и удаленными - находящимися на других доменах.

Прототип этого метода выглядит так:

```
$.getScript(url, [success(data, textStatus)])
```

Первый параметр является обязательным. Второй задает функцию, код которой будет выполнен, если запрос будет успешно выполнен.

Код приложения, запускающего локальный и удаленный JavaScript-сценарии, приведен в листинге 11.26.

Листинг 11.26. Демонстрация метода \$.getScript()

```
<html>
<head>
<title>Метод getScript()</title>
<meta http-equiv="Content-Type" content="text/html;
charset=utf-8" />
</head>
<body>
<div></div>
<button>Запустить локальный скрипт</button>
<button>Запустить удаленный скрипт</button>
<script src="https://code.jquery.com/jquery-3.6.0.js"></script>
<script >
$(function () {
    $("button:eq(0)").click(function() {
        $.getScript("script.js");
    });
    $("button:eq(1)").click(function() {
        $.getScript("http://nit.center/ajax/test.js",
        function(){
            alert('После');
        });
    });
});
</script>
</body>
</html>
```

В случае с удаленным сценарием установлена и функция, которая будет выполнена в случае успешного выполнения сценария. Другими словами, сначала вы должны получить сообщение "Удаленный скрипт" - оно выводится удаленным сценарием, а потом - "После", его генерирует уже наш сценарий.

11.9.5. Получение JSON-данных

Формат JSON очень популярен в Web-программировании и мы просто не могли о нем не упомянуть. JSON используется многими Web-приложениями как стандартный формат обмена данными. В этом разделе мы разработаем сценарий, который получит JSON-данные от PHP-сценария, выполнит их разбор (парсинг) и выведет в удобной форме.

Для начала напишем PHP-сценарий, который выводит JSON-данные (листинг 11.27)

Листинг 11.27. PHP-сценарий json.php

```
<?php
header('Content-Type: text/html; charset=utf-8');

if($_SERVER['HTTP_X_REQUESTED_WITH'] == 'XMLHttpRequest')
{
    print '[{"country":"UK", "capital":"London"},' .
          {"country":"Japan", "capital":"Tokio"},' .
          {"country":"Russia", "capital":"Moscow"}]';
}
?>
```

Обратите внимание на несколько важных моментов. Во-первых, формат JSON подразумевает использование кодировки UTF-8, что мы и указали в заголовке, выводимом функцией `header()`. Понятно, что сам сценарий должен быть сохранен в UTF-8, иначе у вас будет проблема с кодировкой. Если вы используете англоязычные данные,

как показано в примере, проблем у вас не будет, а вот если вы будете использовать символы национальных алфавитов, проблемы с кодировкой будут "на лицо".

Далее мы указываем, что данные должны быть отправлены, только если типа запроса - XMLHttpRequest (запрос XML по протоколу HTTP). Это и есть запрос на JSON-данные. Будь у нас реальный сценарий, то в случае с другим типом запроса он бы отправил данные в другом формате, например, в HTML или как обычный текст.

Теперь приступим к реализации JavaScript-сценария. Сердце нашего сценария - метод `getJSON()`, которому нужно передать имя источника JSON-данных и функцию для их разбора. В качестве источника у нас будет выступать приведенный выше сценарий `json.php`, а для разбора данных будет использоваться следующая функция:

```
$.getJSON("json.php", function(data, textStatus,
jqXHR) {
    var items = [];
    $.each(data, function(i,item){
        items.push(item.country + ' - ' + item.capital);
    });
    $("#res").html(items.join('<br>'));
});
```

Сначала функция создает пустой массив `items[]`. Затем помещает в массив каждый элемент данных из `data`. Аргумент `data` нашей функции как раз содержит полученные от источника JSON-данные. Поле `country` становится свойством `country` объекта `item`, аналогично поле `capital` становится одноименным свойством объекта `item`.

После того, как данные собраны в массив `items[]`, они объединяются методом `join()`, при этом каждый элемент разделяется тегом `
`. Затем все это передается методу `html()` нашей области результата.

Полный код HTML-страницы приведен в листинге 11.28.

Листинг 11.28. Получение и парсинг JSON-данных

```
<html>
<head>
<title>Пример JSON</title>
<meta http-equiv="Content-Type" content="text/html;
charset=utf-8" />

<style type="text/css">
div {
    width:300px; height:150px;
    padding:15px; margin:15px;
    border:2px solid red;
    overflow:auto;
}
</style>

</head>
<body>

<div id="res"></div>
<button id="load">Получить JSON</button>
<button id="clear">Очистить</button>
<script src="https://code.jquery.com/jquery-
3.6.0.js"></script>
<script>
$(function () {
    $("#load").click(function () {
        $.getJSON("json.php", function(data, textStatus, jqXHR) {
            let items = [];
            $.each(data, function(i, item) {
                items.push(item.country + ': ' + item.capital);
            });
            $("#res").html(items.join('<br />'));
        });
    });
    $("#clear").click(function () {
        $("#res").empty();
    });
})</script>
```

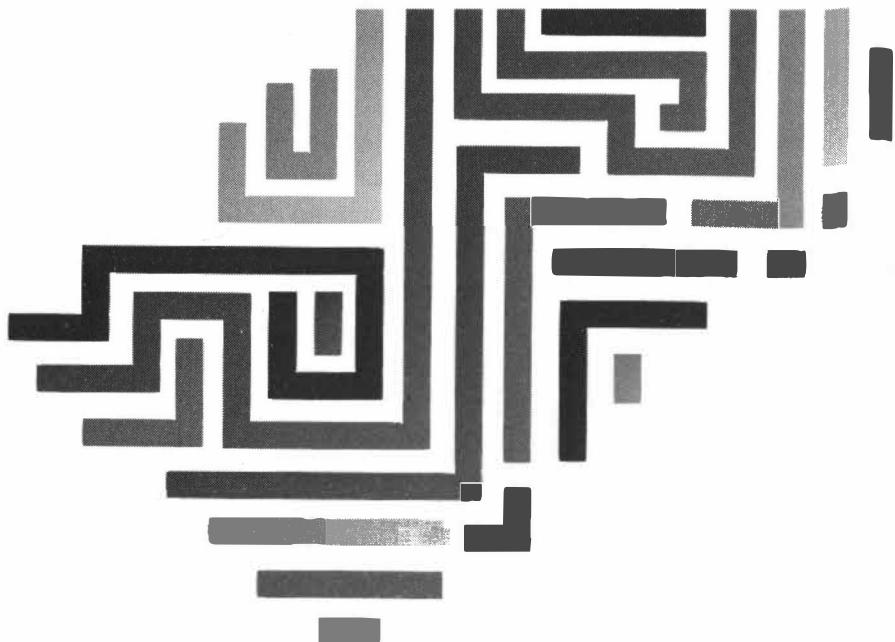
```
    });
});
</script>

</body>
</html>
```

Обратите внимание на кодировку HTML-страницы. Сам HTML-документ должен быть сохранен также в кодировке UTF-8 (мало указать кодировку в заголовке страницы, нужно еще и сохранить сам документ в этой кодировке). В обычном редакторе этого не сделаешь, поэтому я рекомендую использовать бесплатный редактор Notepad2. Он мало того, что работает с различными кодировками, так еще и обеспечивает подсветку синтаксиса.

ГЛАВА 12.

Библиотека jQuery UI



12.1. Знакомство с библиотекой

12.1.1. Что такое jQuery UI

Давайте вернемся на более чем 15 лет назад в прошлое. Примерно в 2006 год. Технология AJAX только зарождалась, вышла 7-ая версия Internet Explorer и... появилась небольшая библиотека под названием jQuery.

Благодаря возможности расширения функционала библиотеки с помощью различных плагинов, она стала очень популярна. Спустя относительно небольшой промежуток времени сообществом jQuery были созданы тысячи плагинов. Конечно, у первых плагинов были недостатки - они были плохо документированы (или вообще не было документации), их API не был стандартизирован. Но плагины были и это радовало.

В сентябре 2007 года появилась новая библиотека, в которую вошли лучшие из созданных плагинов. Эта библиотека стала называться jQuery UI.

Аббревиатура UI, как вы уже догадались, означает User Interface, что отражает функционал библиотеки - она содержит плагины, которые будут полезны для разработки интерфейса пользователя веб-приложения.

Плагины jQuery UI настолько разнообразны, что их можно с успехом использовать на самых разных сайтах - от небольших и простеньких сайтов до целых порталов.

Преимущество библиотеки в том, что API стандартизирован и хорошо задокументирован. Поверьте, очень удобно писать код, когда знаешь, для чего используется тот или иной метод, какие параметры он принимает, что означает та или иная константа.

В отличие от обычных плагинов jQuery, которых тысячи, плагины и утилиты jQuery UI поддерживаются jQuery Foundation. Другими словами, это официальная и одобренная к использованию библиотека.

Точное количество сайтов, использующих jQuery UI, подсчитать довольно сложно: кто-то предпочитает загружать библиотеку из CDN, кто-то загружает на свой сайт, какой-то сайт отображается в Google, какой-то - нет. Все это затрудняет подсчет точного количества.

12.1.2. Что представляют собой плагины jQuery UI

Как говорится, лучше один раз увидеть, чем 100 раз услышать. Поэтому прямо сейчас перейдите на демо-страничку библиотеки:

<https://jqueryui.com/demos/>

Среди виджетов jQuery UI вы найдете:

1. **Accordion** – отображает контент на нескольких панелях для экономии пространства на странице.
2. **Autocomplete** – поле автодополнения. Наверняка вы видели такие поля, типичный пример - поисковая система Google.

Начните вводить начальные символы слова и увидите варианты автоматического дополнения.

3. **Button** – кнопка. В отличие от стандартной кнопки, кнопка jQuery UI выглядит гораздо привлекательнее.
4. **Checkboxradio** – позволяет создавать зависимые и независимые переключатели. Опять-таки, внешний вид более привлекательный, чем использование стандартных HTML-элементов.
5. **Controlgroup** – используется для группировки различных виджетов в один визуальный набор.
6. **Datepicker** – виджет выбора даты (см. рис. 12.1)
7. **Dialog** – виджет, создающий симпатичный диалог, который можно ко всему прочему еще и перетаскивать по всей странице.
8. **Menu** – используется для создания меню сайта.
9. **Progressbar** – довольно полезный компонент для отображения хода процесса.
10. **Selectmenu** – собственный вариант выпадающего списка. Выглядит в одном стиле со всеми остальными элементами jQuery UI.
11. **Slider** – слайдер, который удобно использовать для установки различных числовых (целочисленных) значений.
12. **Spinner** – поле ввода числового значения с кнопками больше/меньше.
13. **Tabs** – виджет, позволяющий создать несколько вкладок с контентом для экономии пространства на странице.
14. **Tooltip** – позволяет создать эффектные подсказки.



Рис. 12.1. Виджет выбора даты

12.1.3. Преимущества jQuery UI

Рассмотрим преимущества использования jQuery UI:

- Непротиворечивый код – вы можете использовать различные виджеты сторонних разработчиков. Может быть, даже найдете те, которые лучше выглядят. Но если вам нужно использовать на одной странице несколько виджетов различных разработчиков, не исключены конфликты. Здесь же вы получаете набор виджетов с непротиворечивым кодом, исключающим всякие конфликты. Все, что вам остается – это просто использовать их, не задумываясь, как они работают и что нужно изменить, чтобы можно было на одной странице использовать более двух виджетов.
- Расширенная поддержка браузеров – над jQuery UI работает огромное сообщество и ее код написан таким образом, что он будет поддерживать большинство самых популярных браузеров (Chrome, Firefox, Internet Explorer, Safari, Opera и т.д.) - в противовес плагинам, которые разрабатываются одним-двумя программистами. Одному человеку сложно выполнить отладку работы сценария во всех браузерах и потом поддерживать код на предмет совместимости с новыми версиями браузеров.
- Код бесплатен и свободно распространяется – все желающие могут абсолютно бесплатно заполучить jQuery UI с <https://>

github.com/jquery/jquery-ui, в отличие от некоторых других платных библиотек.

- Полноценная документация – как уже отмечалось, библиотека хорошо документирована. Описан каждый ее метод, каждый параметр, каждая константа. Получить доступ к документации можно по адресу: <http://api.jqueryui.com/>
- Наличие тем оформления – если вам не нравится, как выглядят некоторые виджеты jQuery UI, это не проблема. Вы можете заполучить другую тему оформления, которая наиболее подходит к дизайну вашего сайта. При этом не нужно менять стили CSS вручную (только в самых сложных случаях), обычно достаточно просто загрузить другую тему оформления с <http://jqueryui.com/themeroller/>. На этом сайте можно не только загрузить тему оформления, но и в реальном времени просмотреть, как будут выглядеть основные виджеты jQuery UI.

12.1.4. Использование jQuery UI

Теперь разберемся, как использовать jQuery UI. Как и в случае с jQuery, вы можете или скачать файлы библиотеки (ссылка была приведена ранее) на свой компьютер или же подключать их из CDN.

В листинге 12.1 приводится "болванка" HTML-страницы, к которой подключена эта библиотека.

Листинг 12.1. Подключение jQuery UI

```
<!doctype html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Болванка</title>
<link rel="stylesheet"
href="https://code.jquery.com/ui/1.12.0/themes/
smoothness/jquery-ui.css">
</head>
<body>
```

```
<!-- Вставьте ваш HTML код здесь -->

<script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
<script src="https://code.jquery.com/ui/1.12.0/jquery-ui.min.js"></script>

<!-- Добавьте ваш JavaScript-код здесь -->
</body>
</html>
```

12.2. Виджеты jQuery UI

12.2.1. Выбор даты

Виджет – это уже готовый элемент интерфейса пользователя. Библиотека jQuery UI содержит целый набор различных элементов пользовательского интерфейса. Например, в этом разделе будет рассмотрен элемент выбора даты. А это означает, что вам никогда не придется разрабатывать его самостоятельно - вы можете использовать уже готовое решение.

Посмотрите на рис. 12.1. На нем изображен виджет DataPicker из библиотеки jQuery UI.

Сейчас и вы научитесь создавать такой же. Представим, что у нас есть поле ввода даты:

```
<p>Дата: <input id="date" type="text" /></p>
```

Ничего особенного, обычное поле с `id="date"`. Вызвать виджет DataPicker можно с помощью метода `datepicker()`:

```
$("#date").datepicker();
```

Вот только, чтобы все заработало как нужно, необходимо подключить библиотеку JQuery UI.

Напоминаю, что при загрузке библиотеки JQuery UI вы можете выбрать одну из тем оформления, которую вы будете использовать. Доступно много тем оформления, к сожалению, нельзя скачать их все сразу, приходится скачать несколько вариантов, а потом извлечь из них необходимые вам CSS-файлы.

Подключить необходимую тему оформления можно как обычный CSS-файл, например:

```
<link type="text/css" href="путь/jquery-ui.css"
rel="stylesheet" />
```

Полный исходный код страницы выбора даты приведен в листинге 12.1. Как выглядит диалог выбора даты, было показано на рис. 12.1.

Листинг 12.2. Использование виджета выбора даты

```
<html>
<head>
<title>Выбор даты: DataPicker</title>
<link type="text/css" href="jquery-ui-1.12.1.custom/
jquery-ui.css" rel="stylesheet" />
</head>
<body>
<p>Дата: <input id="date" type="text" /></p>
<script src="https://code.jquery.com/jquery-3.2.1.js"></
script>
<script src="jquery-ui-1.12.1.custom/jquery-ui.js"></
script>

<script type="text/javascript">
$(function() {
    $("#date").datepicker();
});
</script>
</body>
</html>
```

Как видите, код очень компактен и прост, как и должно быть с библиотеками подобного уровня.

Вот только есть одна проблема. Диалог выбора даты будет на английском языке, а нам бы хотелось видеть русский язык в своих приложениях. Попробуем исправить это, изменив листинг 12.2 так:

Листинг 12.26. Локализация средства выбора даты

```
<html>
<head>
<title>Выбор даты: русифицированная версия</title>
<link
type="text/css" href="https://code.jquery.com/ui/1.12.0/
themes/smoothness/jquery-ui.css" rel="stylesheet" />

</head>
<body>
<p>Дата: <input id="date" type="text" /></p>
<script src="https://code.jquery.com/jquery-3.6.0.min.
js"></script>
<script src="https://code.jquery.com/ui/1.12.0/jquery-ui.
min.js"></script>

<script>

$( function() {
//Сменим язык календаря на русский
    $.datepicker.setDefaults(
        {
            closeText: 'Закрыть',
            prevText: '',
            currentText: 'Сегодня',
            monthNames: ['Январь', 'Февраль', 'Март',
                'Апрель', 'Май', 'Июнь',
                'Июль', 'Август', 'Сентябрь', 'Октябрь',
                'Ноябрь', 'Декабрь'],

```

```

monthNamesShort: ['Янв', 'Фев', 'Мар', 'Апр',
    'Май', 'Июн', 'Июл', 'Авг', 'Сен', 'Окт',
    'Ноя', 'Дек'],
dayNames: ['воскресенье', 'понедельник',
    'вторник', 'среда', 'четверг', 'пятница', 'суббота'],
dayNamesShort: ['вск', 'пнд', 'втр', 'срд',
    'чтв', 'птн', 'сбт'],
dayNamesMin: ['Вс', 'Пн', 'Вт', 'Ср', 'Чт',
    'Пт', 'Сб'],
weekHeader: 'Не',
dateFormat: 'dd.mm.yy',
firstDay: 1,
isRTL: false,
showMonthAfterYear: false,
yearSuffix: ''
}
)
}
);
$(function(){
$("#date").datepicker();
});
</script>
</body>
</html>

```

Вот теперь будет все в порядке (рис. 12.2).



Рис. 12.2. Локализованный вариант выбора даты

12.2.2. Диалоговое окно

С помощью виджета Dialog можно организовать привлекательное диалоговое окно, которое можно перемещать по странице и при необходимости закрыть. Также пользователь может изменять размер такого диалогового окна. Очень полезная штука при выводе различных уведомлений или рекламных блоков.

Организовать такое окно очень просто. Для начала нужно задать текст, который будет использоваться для помещения в диалоговое окно:

```
<div id="dialog" title="Привет!">
    <p>Lorem ipsum dolor sit amet, consectetur adipiscing
    elit. Quisque purus lectus, faucibus ac imperdiet vitae,
    eleifend ut metus. Proin at aliquet purus, nec eleifend
    libero.</p>
```

В данном случае в качестве текста будет использован текст из `<div>` с идентификатором "dialog", а атрибут **title** будет использоваться в качестве заголовка окна.

Далее нужно вызвать метод `dialog()`:

```
$("#dialog").dialog();
```

Собственно, вот и все. Полный исходный код приведен в листинге 12.3, а результат работы метода `dialog()` - на рис. 12.3.

Листинг 12.3. Использование метода `dialog()`

```
<html>
<head>
<title>Виджет dialog</title>
<link rel="stylesheet"
      href="https://code.jquery.com/ui/1.12.0/themes/
      smoothness/jquery-ui.css">
```

```
</head>
<body>
<div id="dialog" title="Привет!">
    <p>Съешь ещё этих мягких французских булок, да выпей чаю.</p>
</div>
<script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
<script src="https://code.jquery.com/ui/1.12.0/jquery-ui.min.js"></script>

<script>
$(function() {
    $("#dialog").dialog();
});
</script>
</body>
</html>
```

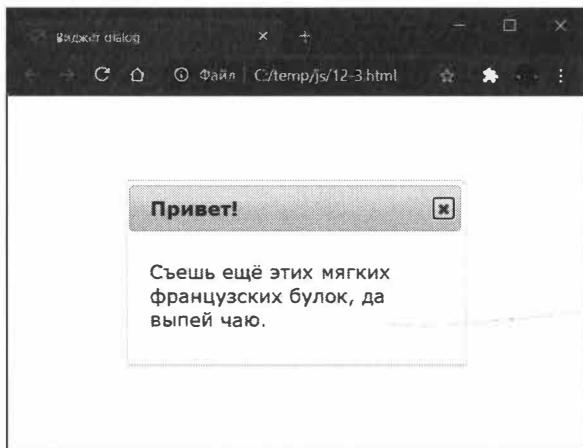


Рис. 12.3. Созданное диалоговое окно

12.2.3. Раскрывающиеся секции

Виджет Accordion позволяет организовать раскрывающиеся секции HTML-кода. Вы не раз видели его на самых разных сайтах – в ка-

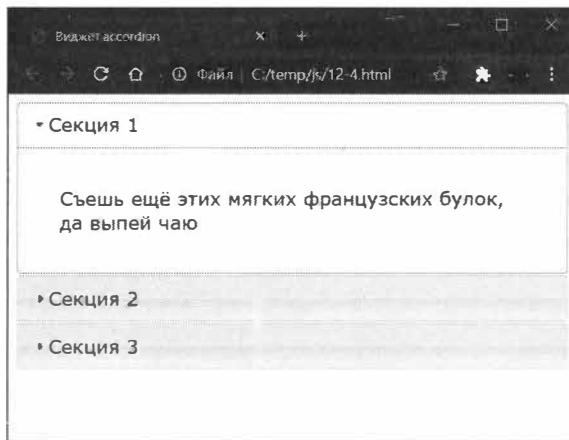


Рис. 12.4. Виджет Accordion в действии

честве списка вопросов/ответов (FAQ), при оформлении заказа (по умолчанию используется в Magento 1.x и других CMS), в карточках продукта и т.д. Чтобы вам было понятно, что мы создаем, взгляните на рис. 12.4. Оказывает организовать такой виджет очень просто.

Первым делом нужно определить `<div>`, содержащий подходящую для Accordion структуру. И именно: заголовки секций должны быть оформлены как `<h3>`, а после заголовка должен следовать `<div>`, содержащий текст раздела (`<div>` обязательно должен быть закрыт!). Внутри `<div>` можно использовать любые HTML-операторы: создавать списки, вставлять картинки, формировать таблицы и т.д.

Вот пример нашего `<div>` с необходимой информацией:

```
<div id="accordion">
  <h3><a href="#">Секция 1</a></h3>
  <div>
    <p>Съешь ещё этих мягких французских булок, да выпей чаю
  </p>
  </div>
  <h3><a href="#">Секция 2</a></h3>
```

```

<div>
  <p>Quisque purus lectus, faucibus ac imperdiet vitae,
eleifend ut metus. </p>
</div>
<h3><a href="#">Секция 3</a></h3>
<div>
  <p>А это просто текст без всякого смысла
</div>

</div>

```

Потом, все, что вам нужно сделать - это вызвать метод **accordion**:

```
$("#accordion").accordion();
```

Собственно, вот и все. Полный код HTML-страницы приведен в листинге 12.4.

Листинг 12.4. Использование виджета Accordion

```

<html>
<head>
<title>Раскрывающиеся секции</title>
<link type="text/css" href="https://code.jquery.
com/ui/1.12.0/themes/smoothness/jquery-ui.css"
rel="stylesheet" />
</head>
<body>
<div id="accordion">
  <h3><a href="#">Секция 1</a></h3>
  <div>
    <p>Съешь ещё этих мягких французских булок, да выпей
чай </p>
  </div>
  <h3><a href="#">Секция 2</a></h3>
  <div>
    <p>Quisque purus lectus, faucibus ac imperdiet vitae,
eleifend ut metus. </p>
  </div>

```

```
<h3><a href="#">Секция 3</a></h3>
<div>
  <p>А это просто текст без всякого смысла
</div>

</div>
<script src="https://code.jquery.com/jquery-3.6.0.min.js"
></script>
<script src="https://code.jquery.com/ui/1.12.0/jquery-ui.
min.js"></script>

<script>
$(function() {
  $("#accordion").accordion();
});
</script>
</body>
</html>
```

12.2.4. Индикатор процесса

Наверняка вы видели индикатор процесса, показывающий ход выполнения какого-то процесса, например, загрузки файла.

Индикатор процесса создается методом `progressbar()`. При изменении значения индикатора (позже я покажу, как это сделать) генерируется событие `progressbarchange`. Когда значение индикатора достигнет 100, будет сгенерировано событие `progressbarcomplete`.

В листинге 12.5 приведен полный код страницы, демонстрирующей управление индикатором процесса.

Примечание. Версии jQuery и jQuery UI существенно отличаются. Здесь нет никакой ошибки. По состоянию на март 2021 года текущей версией jQuery являлась 3.6.0, а jQuery UI – 1.12.

Листинг 12.5. Демонстрация управления индикатором процесса

```
<html>
<head>
<title>Виджет progressbar</title>
<link
type="text/css" href="https://code.jquery.com/ui/1.12.0/
themes/smoothness/jquery-ui.css" rel="stylesheet" />
</head>
<body>
<div id="bar"></div>
<button>Добавить 10</button>
<div id="percent"></div>
<script src="https://code.jquery.com/jquery-3.6.0.min.
js"></script>
<script src="https://code.jquery.com/ui/1.12.0/jquery-ui.
min.js"></script>
<script>
$(function() {
    $("#bar").progressbar({
        value: 0,
        change: function(event, ui) {
            //console.log('Событие ' + event.type);
        },
        complete: function(event, ui) {
            console.log('Событие ' + event.type);
        }
    });
    $("button").click(function() {
        let currentVal = $("#bar").progressbar('option',
        'value');

        if(currentVal<100) {
            currentVal = currentVal + 10;
            $("#percent").text(currentVal + "%");
            $("#bar").progressbar('option', 'value',
            currentVal);
        }
    })
})
```

```
    });
}

</script>
</body>
</html>
```

У нас есть кнопка, увеличивающая значение индикатора процесса на 10. В <div> с идентификатором "result". При нажатии кнопки первым делом мы получаем текущее значение индикатора в переменную currentVal, затем если значение еще не достигло 100, то мы увеличиваем его на 10 (в реальном мире придется увеличивать на столько, сколько реально работы выполнено), выводим в область result и устанавливаем новое значение методом progressbar().

Как только значение индикатора достигнет 100, будет сгенерировано событие *progressbarcomplete* и будет запущен его обработчик - вы увидите диалоговое окно с типом события. А вот если вы хотите (из соображений отладки, конечно) видеть подобное диалоговое окно при каждом изменении значения индикатора, раскомментируйте выделенный жирным оператор. Пример работающего индикатора процесса приведен на рис. 12.5.

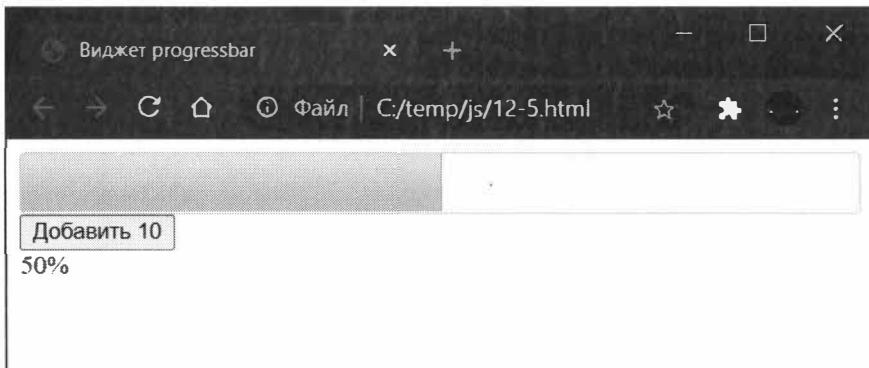


Рис. 12.5. Индикатор процесса

Вообще обработчик события *progressbarchange* можно использовать для чего-то полезного, например, если вы индикатор загрузки отображает процесс установки вашего программного продукта, то можете менять слайды, демонстрируя пользователю особенности вашего продукта. Все мы устанавливали хоть раз в жизни Windows, поэтому, думаю, вы понимаете, о чем я говорю. Конечно, на JavaScript и PHP операционную систему не напишешь, но зато можно написать систему управления контентом и сделать для нее красочный инсталлятор, а в этом как раз вам поможет JQuery UI.

12.2.5. Вкладки

Иногда удобнее представить информацию во вкладках, например, когда вы разрабатываете интерфейс сценария, изменяющий параметры вашего программного продукта. Вместо того чтобы создавать несколько страниц с настройками, вы можете использовать одну страницу, но несколько вкладок (см. рис. 12.6).

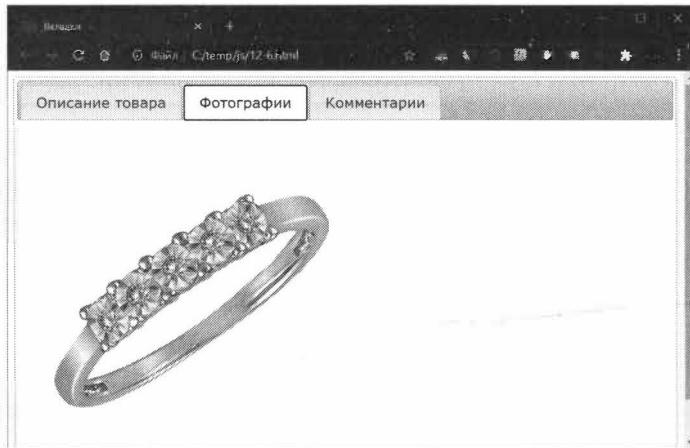


Рис. 12.6. Вкладки, реализованные с помощью JQuery UI

Организовать вкладки очень просто. Для этого нужно создать список, состоящий из заголовков вкладок, а затем в отдельных блоках

(**id** блока должен соответствовать **id** вкладки) описать содержимое каждой вкладки:

```
<div id="tabs">
  <ul>
    <li><a href="#tabs-1">Описание товара</a></li>
    <li><a href="#tabs-2">Фотографии</a></li>
    <li><a href="#tabs-3">Комментарии</a></li>
  </ul>
  <div id="tabs-1">
    <p>Кольцо лаконичного вида привлекает внимание совершенной, сжатой композицией. Узкий обруч нежно-розового, почти зефирного колера в вершине выпрямляется и превращается в дорожку серовато-серебристого цвета, отделанную сияющими бриллиантами и формованным орнаментом, состоящим из лучистых линий и декоративных закрепов в виде зерни.</p>
  </div>
  <div id="tabs-2">
    <p> </p>
  </div>
  <div id="tabs-3">
    <p>Комментариев пока нет</p>
  </div>
</div>
```

После того, как вкладки определены, нужно вызвать метод `tabs()` для родительского блока:

```
$("#tabs").tabs();
```

Полный код страницы с вкладками представлен в листинге 12.6.

Листинг 12.6. Вкладки с помощью JQuery UI

```
<html>
<head>
<title>Вкладки</title>
<link
```

```
type="text/css" href="https://code.jquery.com/ui/1.12.0/
themes/smoothness/jquery-ui.css" rel="stylesheet" />
</head>
<body>

<div id="tabs">
  <ul>
    <li><a href="#tabs-1">Описание товара</a></li>
    <li><a href="#tabs-2">Фотографии</a></li>
    <li><a href="#tabs-3">Комментарии</a></li>
  </ul>
  <div id="tabs-1">
    <p>Кольцо лаконичного вида привлекает внимание
совершенной, сжатой композицией. Узкий обруч
нежно-розового, почти зефирного колера в вершине
выпрямляется и превращается в дорожку серовато-
серебристого цвета, отделанную сияющими бриллиантами и
формованным орнаментом, состоящим из лучистых линий и
декоративных закрепов в виде зерни.</p>
  </div>
  <div id="tabs-2">
    <p> </p>
  </div>
  <div id="tabs-3">
    <p>Комментариев пока нет</p>
  </div>
</div>
<script src="https://code.jquery.com/jquery-3.6.0.min.
js"></script>
<script src="https://code.jquery.com/ui/1.12.0/jquery-ui.
min.js"></script>

<script>
$(function() {
  $("#tabs").tabs();
});
</script>
</body>
</html>
```

12.2.6. Автозавершение

Еще одна полезная функция вашего будущего программного продукта, которую можно легко реализовать средствами JQuery UI - это автозавершение слов. Наверняка вы видели списки, в которые можно ввести часть слова или фразы, как сразу же открывается список и начинает предлагать доступные варианты. Сейчас мы реализуем подобную функциональность с помощью метода autocomplete().

Данному методу нужно передать в качестве параметра список, определяющий доступные пользователю варианты. Например:

```
$("#cars").autocomplete({  
    source: availableCars  
});
```

Обратите внимание, как хитро задается список. Не просто нужно указать имя списка, а передать его как параметр **source**. А вот пример самого списка:

```
var availableCars = ["Acura", "Alfa", "Romeo", "Audi",  
"Bentley", "BMW", "Buick", "BYD", "Cadillac", "Caterham",  
"Chery", "Chevrolet", "Chrysler", "Citroen", "Daewoo",  
"Daihatsu", "Dodge", "Ferrari", "FIAT", "Ford", "Geely",  
"GMC", "Great", "Wall", "Honda", "Hummer", "Hyundai",  
"Infiniti", "Isuzu", "Jaguar", "Jeep", "KIA", "Lancia",  
"Land", "Rover", "Lexus", "Lifan", "Lincoln", "Lotus",  
"Maserati", "Mazda", "Mercedes-Benz", "Mercury", "MG",  
"MINI", "Mitsubishi", "Nissan", "Opel", "Peugeot",  
"Plymouth", "Pontiac", "Porsche", "Renault", "Rover",  
"Saab", "Saturn", "Scion", "SEAT", "Skoda", "Smart",  
"SsangYong", "Subaru", "Suzuki", "Toyota", "Volkswagen",  
"Volvo"];
```

А вот как определяется само поле для ввода:

```
<input id="cars" type="text" />
```

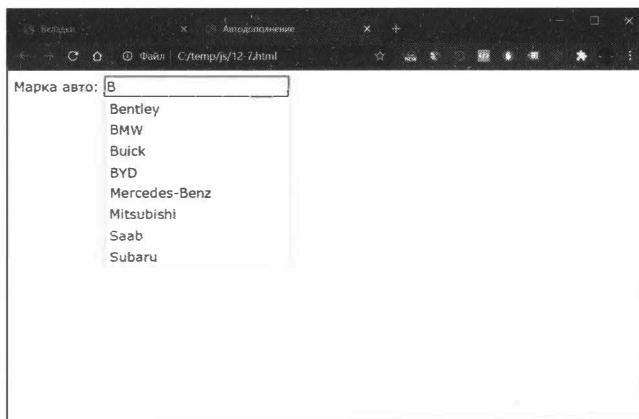


Рис. 12.7. Работающий список автодополнения

Пример работающего списка приведен на рис. 12.7.

Полный код страницы, демонстрирующей, как работает автодополнение, приведен в листинге 12.7.

Листинг 12.7. Демонстрация работы автодополнения списка

```
<html>
<head>
<title>Автодополнение списка</title>
<link
type="text/css" href="https://code.jquery.com/ui/1.12.0/
themes/smoothness/jquery-ui.css" rel="stylesheet" />
</head>
<body>
<div class="ui-widget">
  <label for="cars">Марка авто: </label>
  <input id="cars" type="text" />
</div>
<script src="https://code.jquery.com/jquery-3.6.0.min.
js"></script>
<script src="https://code.jquery.com/ui/1.12.0/jquery-ui.
min.js"></script>
```

```
<script>
$(function(){

    var availableCars = ["Acura", "Alfa", "Romeo", "Audi",
    "Bentley", "BMW", "Buick", "BYD", "Cadillac", "Caterham",
    "Chery", "Chevrolet", "Chrysler", "Citroen", "Daewoo",
    "Daihatsu", "Dodge", "Ferrari", "FIAT", "Ford", "Geely",
    "GMC", "Great", "Wall", "Honda", "Hummer", "Hyundai",
    "Infiniti", "Isuzu", "Jaguar", "Jeep", "KIA", "Lancia",
    "Land", "Rover", "Lexus", "Lifan", "Lincoln", "Lotus",
    "Maserati", "Mazda", "Mercedes-Benz", "Mercury", "MG",
    "MINI", "Mitsubishi", "Nissan", "Opel", "Peugeot",
    "Plymouth", "Pontiac", "Porsche", "Renault", "Rover",
    "Saab", "Saturn", "Scion", "SEAT", "Skoda", "Smart",
    "SsangYong", "Subaru", "Suzuki", "Toyota", "Volkswagen",
    "Volvo"];

    $("#cars").autocomplete({
        source: availableCars
    });
});

</script>

</body>
</html>
```

12.2.7. Кнопка

Описание библиотеки JQuery UI было бы не полным, если бы мы не рассмотрели, какие она умеет формировать кнопки. Казалось бы, нужно было с кнопок и начать, поскольку кнопки всегда считаются самыми примитивными виджетами, однако хотелось сразу начать с чего-то более интересного и полезного. Тем более, кнопки JQuery UI мне кажутся немного вычурными (рис. 12.8). С одной стороны, симпатично, с другой они впишутся далеко не в каждый дизайн, поэтому придется экспериментировать с CSS-кнопки или же с другой темой оформления для всего JQuery UI.

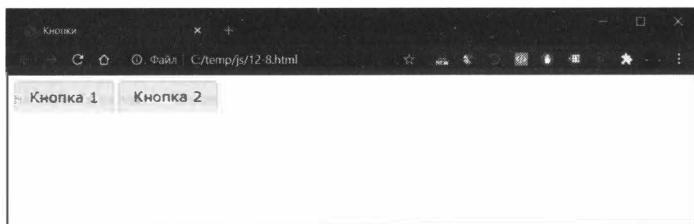


Рис. 12.8. Виджет Button

Рассмотрим код страницы, изображенной на рис. 12.8 (листинг 12.8).

Листинг 12.8. Код страницы с кнопками

```
<html>
<head>
<title>Кнопки jQuery UI</title>
<link
type="text/css" href="https://code.jquery.com/ui/1.12.0/
themes/smoothness/jquery-ui.css" rel="stylesheet" />
</head>
<body>
    <button id="button1">Кнопка 1</button>
    <button id="button2">Кнопка 2</button>
    <script src="https://code.jquery.com/jquery-3.6.0.min.
js"></script>
    <script src="https://code.jquery.com/ui/1.12.0/jquery-
ui.min.js"></script>
    <script>
        $(function () {
            $("#button1").button().click(function (e) {
                console.log("Нажата кнопка 1");
            });
            $("#button2").button().click(function (e) {
                console.log("Нажата кнопка 2");
            });
        });
    </script>
```

```
</script>
</body>
</html>
```

Нужно отметить, что мало просто подключить библиотеку JQuery UI, чтобы все кнопки превратились в кнопки, показанные на рис. 12.8. Как видно из кода, представленного в листинге 12.8, для наших кнопок вызывался метод `button()`. Именно он и выполняет превращение обычной кнопки в UI-кнопку. Если его не вызвать, то кнопка будет выглядеть как обычно.

12.2.8. Меню

Виджет `menu` позволяет создать довольно неплохое меню (рис. 12.9). Элементы меню описываются как элементы маркированного списка ``. Код, отображающий меню, изображенное на рис. 12.9, приведен в листинге 12.9. Код взят из официальной документации jQuery и немного переделан.

Листинг 12.9. Виджет `menu`

```
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1">
  <title>Example 11.8</title>
  <link type="text/css"
    href="jquery-ui-1.12.1.custom/jquery-ui.css"
    rel="stylesheet" />
  <style>
    .ui-menu { width: 150px; }
  </style>
</head>
<body>
<ul id="menu">
```

```
<li class="ui-state-disabled"><div>Toys (n/a)</div></li>
<li><div>Books</div></li>
<li><div>Clothing</div></li>
<li><div>Electronics</div>
  <ul>
    <li class="ui-state-disabled"><div>Home
Entertainment</div></li>
    <li><div>Car Hifi</div></li>
    <li><div>Utilities</div></li>
  </ul>
</li>
<li><div>Movies</div></li>
<li><div>Music</div>
  <ul>
    <li><div>Rock</div>
      <ul>
        <li><div>Alternative</div></li>
        <li><div>Classic</div></li>
      </ul>
    </li>
    <li><div>Jazz</div>
      <ul>
        <li><div>Free jazz</div></li>
        <li><div>Big Band</div></li>
        <li><div>Modern</div></li>
      </ul>
    </li>
    <li><div>Pop</div></li>
  </ul>
</li>
<li class="ui-state-disabled"><div>Specials (n/a)</div></li>
</ul>
<script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
<script src="https://code.jquery.com/ui/1.12.0/jquery-ui.min.js"></script>

<script>
$( function() {
```

```
$( "#menu" ).menu();  
} );  
</script>  
</body>  
</html>
```

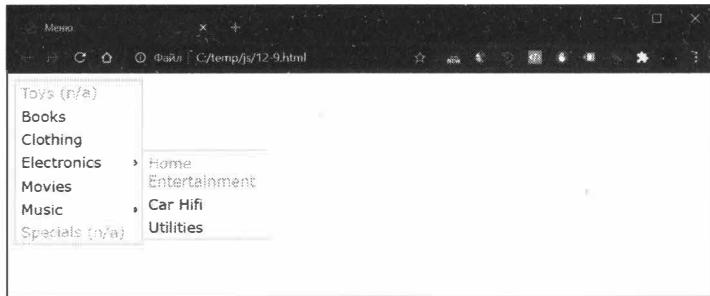


Рис. 12.9. Меню средствами jQuery UI

В принципе ничего сложного, но это до того момента, как вам понадобится горизонтальное меню. Тогда придется погружаться в дебри JavaScript и CSS, поскольку в официальном руководстве об этом ни слова. В листинге 12.10 приведен пример создания горизонтального меню (рис. 12.10).

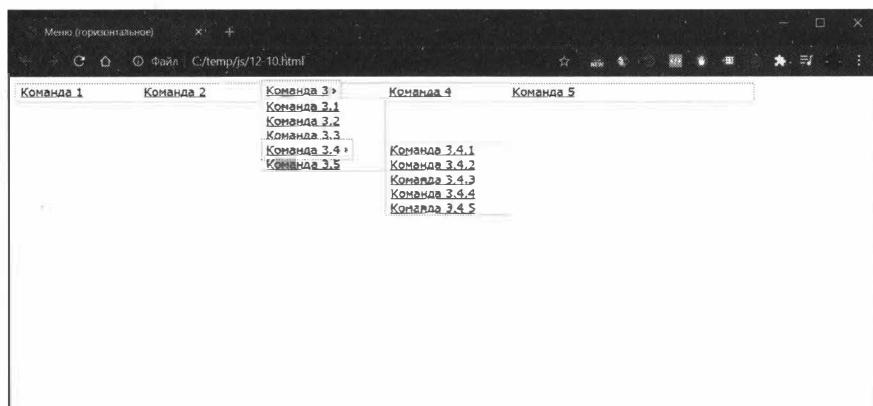


Рис. 12.10. Горизонтальное меню средствами jQuery UI

Листинг 12.10. Горизонтальное меню средствами jQuery UI

```
<html>
<head>
<title>Меню (горизонтальное)</title>
<link
type="text/css" href="https://code.jquery.com/ui/1.12.0/
themes/smoothness/jquery-ui.css" rel="stylesheet" />

<style>
.ui-menu { width: 150px; font-size: 14px; min-height:
22px; }
ul#horMenu { width: 900px; }
ul#horMenu > li { width: 150px; float: left; }
</style>

</head>
<body>
<ul id="horMenu">
<li><a href="#">Команда 1</a></li>
<li><a href="#">Команда 2</a></li>
<li><a href="#">Команда 3</a>
<ul>
<li><a href="#">Команда 3.1</a></li>
<li><a href="#">Команда 3.2</a></li>
<li><a href="#">Команда 3.3</a></li>
<li><a href="#">Команда 3.4</a>
<ul>
<li><a href="#">Команда 3.4.1</a></li>
<li><a href="#">Команда 3.4.2</a></li>
<li><a href="#">Команда 3.4.3</a></li>
<li><a href="#">Команда 3.4.4</a></li>
<li><a href="#">Команда 3.4.5</a></li>
</ul>
</li>
<li><a href="#">Команда 3.5</a></li>
</ul>
</li>
<li><a href="#">Команда 4</a></li>
```

```
<li><a href="#">Команда 5</a></li>
</ul>

<script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
<script src="https://code.jquery.com/ui/1.12.0/jquery-ui.min.js"></script>
<script>
    $(document).ready(function() {
        $("#horMenu").menu({ position: { using:
setSubMenu } });
        $("#horMenu > li > a > span.ui-icon-carat-d-e").removeClass("ui-icon-carat-1-e").addClass("ui-icon-carat-1-s");

        function setSubMenu(position, elements) {
            var options = {
                of: elements.target.element
            };

            if (elements.element.element.parent().parent().parent().attr("id") === "horMenu") {
                options.my = "center top";
                options.at = "center bottom";
            } else {
                options.my = "left top";
                options.at = "right top";
            }

            elements.element.element.position(options);
        };
    });
</script>

</body>
</html>
```

12.2.9. Эффектная подсказка ToolTip

Стандартные подсказки для элементов графического интерфейса, которые генерирует браузер, выглядят не очень красиво. Но jQuery UI может сделать их гораздо красивее, а внешний вид подсказки легко модифицировать с помощью CSS (чего нельзя сделать для стандартных подсказок) - вручную или путем загрузки файла темы jQuery UI.

Для создания красивой подсказки нужно использовать виджет `tooltip`, пример использования которого приведен в листинге 12.11.

Листинг 12.11. Создание подсказки

```
<html>
<head>
<title>Подсказка для элемента</title>
<link
type="text/css" href="https://code.jquery.com/ui/1.12.0/
themes/smoothness/jquery-ui.css" rel="stylesheet" />
<style>
label {
    display: inline-block;
    width: 5em;
}
</style>
</head>
<body>
<p><label for="age">Возраст:</label>
<input id="age" title="Данные не будут переданы третьей
стороне"></p>
<script src="https://code.jquery.com/jquery-3.6.0.min.
js"></script>
<script src="https://code.jquery.com/ui/1.12.0/jquery-ui.
min.js"></script>
<script type="text/javascript">
$( function() {
    $( document ).tooltip();
```

```
    } );
</script>

</body>
</html>
```

Конечный результат, то есть более красивая подсказка для поля ввода, показан на рис. 12.11.

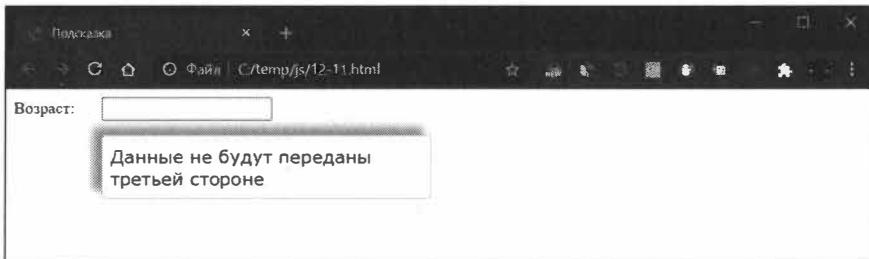


Рис. 12.11. Подсказка, созданная с помощью jQuery UI Tooltip

12.2.10. Ползунок для ввода целых значений

Для установки целочисленного значения, например, возраста, часто используют элемент `<input>` типа `number` (`<input type="number">`). Но почему бы не освежить интерфейс и не использовать внешне привлекательный ползунок (в терминологии jQuery UI – слайдер)? Тем более что создать такой слайдер достаточно просто, используя jQuery UI (лист. 12.12)

Листинг 12.12. Два слайдера

```
<html>
<head>
<title>Ползунок</title>
<meta http-equiv="Content-Type" content="text/html;
charset=utf-8" />

<link
```

```
type="text/css" href="https://code.jquery.com/ui/1.12.0/
themes/smoothness/jquery-ui.css" rel="stylesheet" />

</head>
<body>
<div id="slider"></div>
<p>
<div id="slider2"></div>
<script src="https://code.jquery.com/jquery-3.6.0.min.
js"></script>
<script src="https://code.jquery.com/ui/1.12.0/jquery-ui.
min.js"></script>

<script>
$( function() {
  $( "#slider" ).slider({
    range: true,
    values: [ 0, 100 ]
  });
}

$( function() {
  $( "#slider2" ).slider({
    range: "max",
    value: 100
  });
}

</script>
</body>
</html>
```

Мы только что создали два слайдера. Первый слайдер позволяет выбрать диапазон значений, второй - выбрать одно значение.

Но слайдер - это такой компонент, который просто перемещать неинтересно. Давайте напишем приложение, позволяющее изменить размер шрифта `<textarea>` с помощью слайдера.

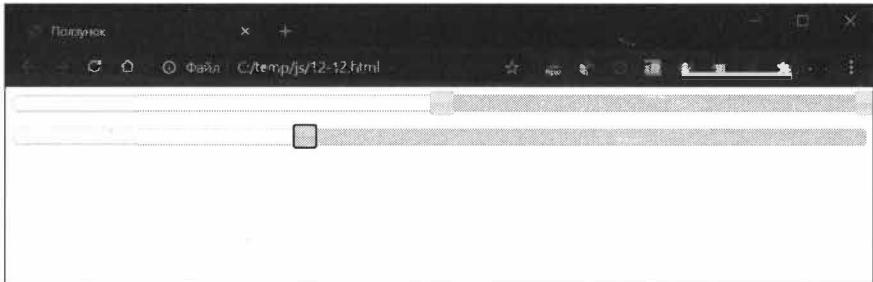


Рис. 12.12. Два ползунка

Рассмотрим следующий код:

```
$( function() {
    $( "#slider" ).slider({
        range: "min",
        value: 1,
        min: 0.5,
        max: 2.5,
        step: 0.1,
        slide: function( event, ui ) {
            $( "#message" ).css( "font-size", ui.value + "em"
        );
    }
});
});
```

Здесь мы выбираем элемент с id "slider" и превращаем его в слайдер. Начальное значение слайдера - 1. Минимальное - 0.5 (min), максимальное - 2.5 (max), шаг изменения значения - 0.1 (step).

С помощью свойства **slide** мы устанавливаем обработчик события перемещения. Данная функция ищет элемент "message" и с помощью метода **css()** изменяет размер шрифта. Размер шрифта будет храниться в **ui.value**.

Полный код примера приведен в листинге 12.13, а результат - на рис. 12.13.

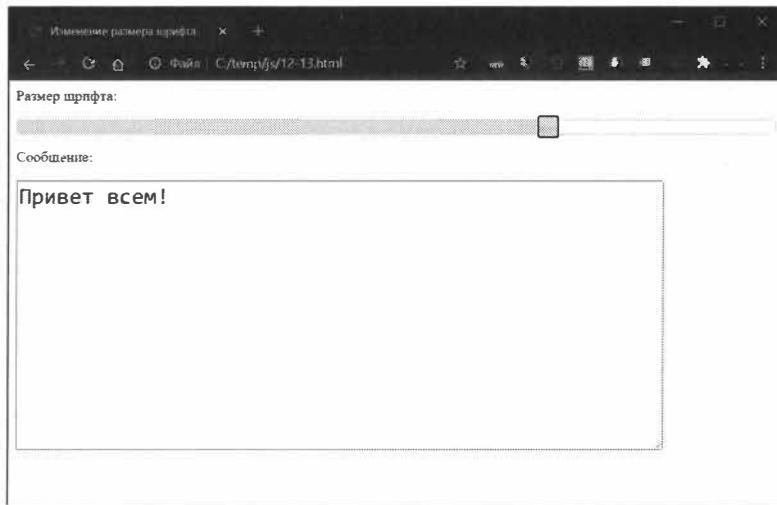


Рис. 12.13. Изменение размера шрифта <textarea> с помощью слайдера

Листинг 12.13. Изменение размера шрифта с помощью слайдера

```
<html>
<head>
<title>Изменение размера шрифта</title>
<link
type="text/css" href="https://code.jquery.com/ui/1.12.0/
themes/smoothness/jquery-ui.css" rel="stylesheet" />
</head>
<body>
Размер шрифта: <p>
<div id="slider"></div>
<p>Сообщение:<br>
<p><textarea id="message" rows="10" cols="50">
Привет всем!
</textarea>
<script src="https://code.jquery.com/jquery-3.6.0.min.
js"></script>
<script src="https://code.jquery.com/ui/1.12.0/jquery-ui.
min.js"></script>
```

```
<script>
$( function() {
    $( "#slider" ).slider({
        range: "min",
        value: 1,
        min: 0.5,
        max: 2.5,
        step: 0.1,
        slide: function( event, ui ) {
            $( "#message" ).css( "font-size", ui.value + "em"
        }
    });
} );
</script>

</body>
</html>
```

На этом все, но мы рассмотрели не все виджеты. Если вы заинтересовались, получить информацию о дополнительных виджетах, а также испытать их в действии, вы всегда можете по адресу:

<http://jqueryui.com/demos/>

12.3. Анимационные эффекты jQuery UI

Библиотека jQuery UI содержит 15 встроенных анимационных эффектов, позволяющие сделать ваш интерфейс более привлекательным. Когда-то веб-мастера подключали jQuery даже не ради ее виджетов, а просто ради анимационных эффектов.

Конечно, не все эффекты jQuery UI имеют практическую ценность. Вы можете, например, разорвать элемент на 50 частей (да, есть такой эффект!), но это не означает, что вы должны это делать. Скорее все-

го, такой эффект вызовет только раздражение у пользователя. Подумайте над этим, прежде чем применять тот или иной эффект.

Далее будут рассмотрены только полезные эффекты, которые вы сможете применять на практике.

12.3.1. Как применить эффект

Рассмотрим, как можно применить эффект к элементу страницы:

```
<style>
div {
    background: black;
    height: 100px;
    width: 100px;
}
</style>
<div></div>

<script>
$( "div" ).effect( "shake" );
</script>
```

У нас есть стиль, устанавливающий фон (красный) и размеры <div> - 100x100 пикселей, то есть, по сути, создающий черный квадрат. После чего к этому черному квадрату мы применяем эффект **shake** (дрожание).

Результат приводить не стану, поскольку статическая иллюстрация в книге не передаст анимационный эффект, зато в листинге 12.14 приведен полный код данного примера.

Листинг 12.14. Применения эффекта shake (дрожание)

```
<html>
<head>
<title>Эффект shake (дрожание)</title>
```

```
<link
type="text/css" href="https://code.jquery.com/ui/1.12.0/
themes/smoothness/jquery-ui.css" rel="stylesheet" />

<style>
div {
background: red;
height: 110px;
width: 110px;
}
</style>

</head>
<body>

<div></div>

<script src="https://code.jquery.com/jquery-3.6.0.min.
js"></script>
<script src="https://code.jquery.com/ui/1.12.0/jquery-ui.
min.js"></script>

<script>
 $("div").effect("shake");
</script>

</body>
</html>
```

12.3.2. Эффекты и их параметры

Всего доступно 15 эффектов:

1. blind – "Ослепление";
2. bounce – "Отскок";
3. clip – "Отсечение";
4. drop – "Падение";

5. explode – "Взрыв" (да, этот тот самый эффект, речь о котором шла в начале раздела);
6. fade – "Затухание";
7. fold – "Складка";
8. highlight – "Освещение";
9. puff – "Рассеивание";
10. pulsate – "Пульсирование";
11. scale – "Масштабирование" (позволяет увеличить или уменьшить элемент страницы);
12. shake – "Дрожание";
13. size – "Калибровка";
14. slide – "Скольжение";
15. transfer – "Переход".

В принципе, назначение эффекта понятно из его названия. Исключение составляет разве что эффект **size** (калибровка). С помощью опции **to:** и включенных в нее опций **width:ширина_в_пикселях** и **height:высота_в_пикселях** вы можете задать размеры, до которых необходимо "откалибровать" текущий элемент.

Также не совсем понятен эффект **transfer**. С помощью опции **to:id_или_class_элемента** вы можете указать элемент, в который перейдет текущий элемент. С помощью опции **className:имя_класса** вы можете оформить эффект перехода с помощью CSS.

У каждого эффекта есть параметры. Некоторые эффекты вроде **shake** можно использовать с параметрами по умолчанию, то есть, не указывая параметры при вызове эффекта - визуально все будет выглядеть приемлемо. Некоторые же эффекты (тот же **size**) без задания параметров использовать не получится, поскольку визуальный эффект будет отличаться от ожидаемого.

Задать параметры для эффекта можно так:

```
$( "div" ).effect( "shake", {  
    times: 10,  
    distance: 100  
});
```

Здесь мы определяем два параметра - **times** (количество колебаний элемента) и **distance** (насколько элемент будет отклоняться при колебаниях, по сути, задает амплитуду колебания).

Откуда узнать, какие параметры у эффекта есть и что они означают? В этом вам поможет официальная документация. Чтобы получить список параметров (и их назначение) эффекта, введите URL:

<https://api.jqueryui.com/{название}-effect/>

Например, для эффекта **shake** URL будет выглядеть так:

<https://api.jqueryui.com/shake-effect/>

Дополнительную информацию об эффектах вы можете получить по адресу:

<https://api.jqueryui.com/category/effects/>

Если вы попробуете задать приведенные ранее параметры эффекта **shake** в листинге 12.14, то вы заметите, что эффект выполняется слишком быстро. Поэтому пора познакомиться со вторым параметром метода **effect()** - **duration**, то есть продолжительность эффекта. Продолжительность задается в миллисекундах, 1000 миллисекунд - это одна секунда:

```
$( "div" ).effect( "shake", {  
    times: 10,  
    distance: 100  
}, 5000 );
```

Приведенный код заставит наш квадрат дрожать 5 секунд.

У метода effect() есть третий параметр, позволяющий задать функцию-обработчик, которая будет вызвана по окончанию анимации. Пример:

```
$( "div" ).effect( "shake", {
  times: 10,
  distance: 100
}, 5000, function() {
  $( this ).css( "background", "red" );
});
```

Это все хорошо, но такой код достаточно сложен для восприятия и написания - новичок может допустить много ошибок при его написании, а если код писал не он, то новичку будет сложно его читать. Именно поэтому вы можете использовать параметры **duration** и **complete** самого эффекта. Первый задает продолжительность анимации (аналогично второму параметру метода effect()), а второй - задает обработчик, который будет вызван по окончанию анимации:

```
$( "div" ).effect({
  effect: "shake",
  times: 10,
  distance: 100,
  duration: 3000,
  complete: function() {
    $( this ).css( "background", "red" );
  }
});
```

Такой код гораздо проще для восприятия, чем первоначальный вариант. Но в этом случае обратите внимание на то, как мы задаем название эффекта. Мы должны его передать в качестве значения параметра **effect**.

12.3.3. Демонстрация всех эффектов сразу

Теоретически, вам достаточно знать один метод `effect()` и его будет достаточно в большинстве случаев. На этом данную главу можно было бы и закончить, но тогда она вряд ли была интересной и ценной с практической точки зрения.

Сейчас мы напишем сценарий, демонстрирующий использование всех эффектов, для вызова которых не нужно задавать каких-либо дополнительных параметров. Сценарий будет работать достаточно просто. В нем будет список, позволяющий выбрать эффект и две кнопки - **Скрыть** и **Показать**. Первая скрывает элемент (`<div>`), вторая - показывает. Сокрытие/отображение объекта будет сопровождаться анимационным эффектом. Сам анимационный эффект мы будем задавать не в методе `effect()`, а в методах `hide()` и `show()`. Первый скрывает элемент, второй - отображает его. Как раз увидите, как можно использовать анимационные эффекты в этих методах. Собственно, сам JavaScript-код несложный и выглядит так:

```
<script>
$(document).ready(function() {
    $("#but1").click(function() {
        $("#testcontainer").hide($("#effect1") .
val(), {}, 1000);
    });
    $("#but2").click(function() {
        $("#testcontainer").show($("#effect1") .
val(), {}, 1000);
    });
});
```

Мы видим обработчики двух кнопок - `but1` и `but2`. При нажатии первой кнопки к `<div>` с id "testcontainer" применяется метод `hide`, при нажатии второй кнопки к этому `<div>` применяется метод `show()`.

При этом список эффектов выглядит так:

```
<select id="effect1" name="effect">
  <option value="blind">Blind</option>
  <option value="bounce">Bounce</option>
  <option value="clip">Clip</option>
  <option value="drop">Drop</option>
  <option value="explode">Explode</option>
  <option value="fade">Fade</option>
  <option value="fold">Fold</option>
  <option value="highlight">Highlight</option>
  <option value="puff">Puff</option>
  <option value="pulsate">Pulsate</option>
  <option value="shake">Shake</option>
  <option value="slide">Slide</option>
</select>
```

Полный код странички с нашим сценарием приведен в листинге 12.15.

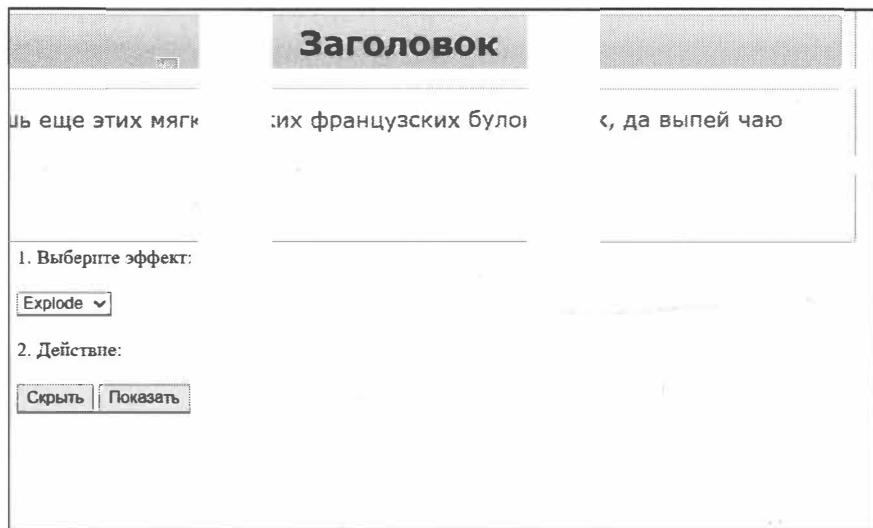


Рис. 12.14. Эффект `explode` в действии (сложно "сфотографировать" анимацию, поэтому просим прощение за такой скриншот)

Листинг 12.15. Демонстрация эффектов, не требующих задания дополнительных параметров

```
<html>
<head>
<title>Эффекты jQuery UI</title>
<link type="text/css" href="https://code.jquery.
com/ui/1.12.0/themes/smoothness/jquery-ui.css"
rel="stylesheet" />

<style type="text/css">
#mycontainer
{
padding:10px;
width:550px;
height:150px;
border:1px solid lightgrey;
}
#mycontainer h2
{
margin-top:0px;
margin-bottom:15px;
text-align:center;
padding:5px;
}
#mycontainer p
{
margin-top:0px;
margin-bottom:0px;
}
</style>

</head>
<body>

<div id="mycontainer" class="ui-widget ui-corner-all">
<h2>Заголовок</h2>
<p>Съешь еще этих мягких французских булок, да выпей
чая</p>
</div>
```

```
<br />
1. Выберите эффект:<br /><br />
<select id="effect1" name="effect">
<option value="blind">Blind</option>
<option value="bounce">Bounce</option>
<option value="clip">Clip</option>
<option value="drop">Drop</option>
<option value="explode">Explode</option>
<option value="fade">Fade</option>
<option value="fold">Fold</option>
<option value="highlight">Highlight</option>
<option value="puff">Puff</option>
<option value="pulsate">Pulsate</option>
<option value="shake">Shake</option>
<option value="slide">Slide</option>
</select>
<br /><br />
2. Действие:<br /><br />
<input id="but1" type="button" value="Скрыть" />
<input id="but2" type="button" value="Показать" />
<br /><br />
<script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
<script src="https://code.jquery.com/ui/1.12.0/jquery-ui.min.js"></script>

<script>
$(document).ready(function() {
    $("#but1").click(function() {
        $("#mycontainer").hide($("#effect1").val(), {}, 1000);
    });
    $("#but2").click(function() {
        $("#mycontainer").show($("#effect1").val(), {}, 1000);
    });
});
</script>

</body>
</html>
```

Все это хорошо, но у нас осталось еще три эффекта - **scale**, **size** и **transfer**. Все они требуют указания дополнительных параметров, и мы их не можем не рассмотреть. Тем более, они считаются более сложными, поскольку не работают без указания параметров.

У нас будет список выбора эффектов, а также три <div>, содержащие параметры каждого эффекта. При выборе эффекта нужно отобразить <div>, содержащий его параметры, а также скрыть остальные <div> - с параметрами других эффектов. За это будет отвечать следующий код:

```
$("#effect1").change(function() {
    if ($("#effect1").val() === "scale") {
        $("#scalecont").css("display", "block");
        $("#transfercont").css("display", "none");
        $("#sizecont").css("display", "none");
    } else if ($("#effect1").val() === "transfer") {
        $("#transfercont").css("display", "block");
        $("#scalecont").css("display", "none");
        $("#sizecont").css("display", "none");
    } else if ($("#effect1").val() === "size") {
        $("#sizecont").css("display", "block");
        $("#transfercont").css("display", "none");
        $("#scalecont").css("display", "none");
    }
});
```

Список **effect** выглядит так:

```
<select id="effect1" name="effect">
<option value="scale">Масштаб</option>
<option value="size">Размер</option>
<option value="transfer">Трансфер</option>
</select>
```

А вот как описаны <div> с параметрами каждого эффекта:

```
<div id="scalecont">
1.1. Масштаб, %:
```

```

<input id="scalepercent" type="text" value="50"/>
</div>
<div id="transfercont">
1.1. κ:
<input id="trans1" type="text" value="#but1"/><br /><br />
1.2. class:
<input id="trans2" type="text" value="transfer-effect"/>
</div>
<div id="sizecont">
1.1. Ширина:
<input id="size1" type="text" value="200"/>
<br /><br />
1.2. Высота:
<input id="size2" type="text" value="200"/>
</div>

```

Обратите внимание на имена (id) блоков и сопоставьте их с кодом.

Осталось рассмотреть код применения эффектов. Он будет стандартным, за тем исключением, что для каждого эффекта мы указываем параметры, введенные пользователем в соответствующие поля ввода:

```

$( "#but1" ).click(function() {
    var options={};
    if ($("#effect1").val() === "scale") {
        var options={percent:50};
        options.percent=$("#scalepercent").val();
    } else if ($("#effect1").val() === "transfer") {
        var options={to:"#but1", className:"transfer-effect"};
        options.to=$("#trans1").val();
        options.className=$("#trans2").val();
    } else if ($("#effect1").val() === "size") {
        var options={to:{width:200,height:200}};
        options.to.width=$("#size1").val();
        options.to.height=$("#size2").val();
    }
    $("#testcontainer").effect($("#effect1").val(),
options ,1000);
});

```

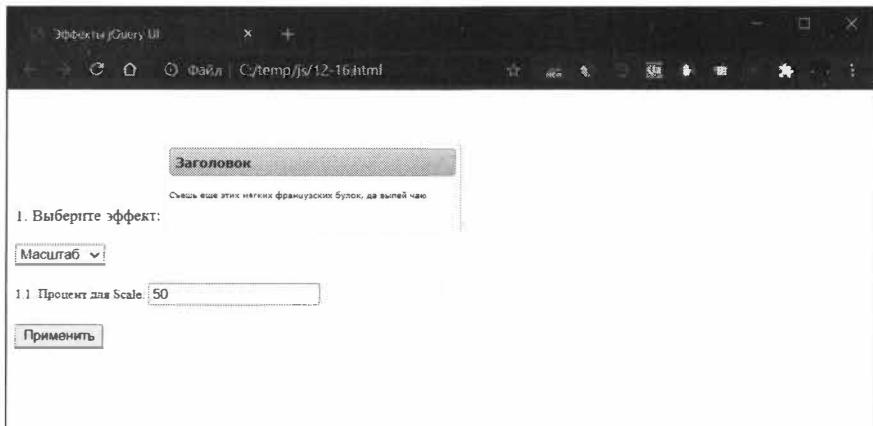


Рис. 12.15. Эффект Scale

На рис. 12.15 изображен эффект **Scale**. Видно, что мы установили масштаб 50% и наш тестовый блок стал меньше. На рис. 12.16 изображен эффект **Size** - здесь мы установили новые размеры (300x200) и блок приобрел новые размеры.

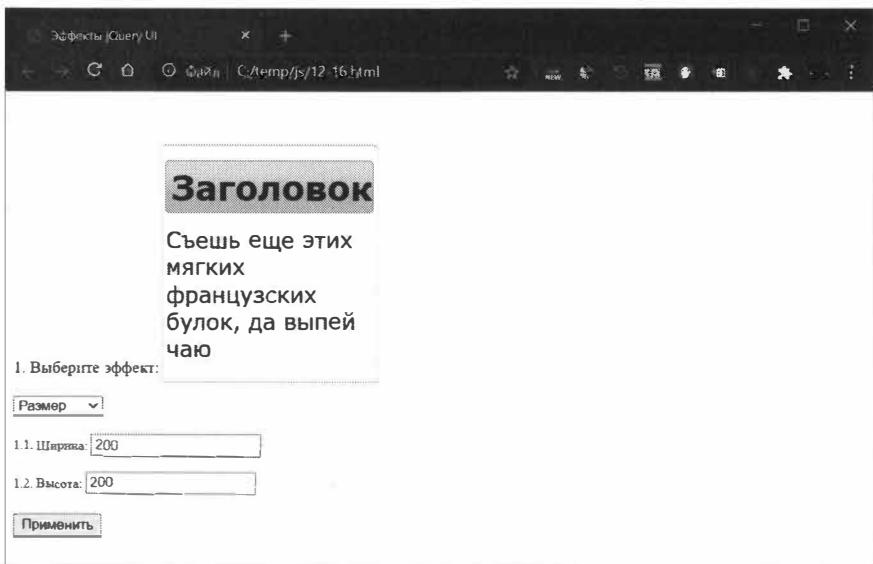


Рис. 12.16. Эффект Size

Полный код примера представлен в листинге 12.16.

Листинг 12.16. Эффекты, требующие установки параметров

```
<html>
<head>
<title>Эффекты jQuery UI</title>
<meta http-equiv="Content-Type" content="text/html;
charset=utf-8" />

<link type="text/css" href="https://code.jquery.
com/ui/1.12.0/themes/smoothness/jquery-ui.css"
rel="stylesheet" />

<style type="text/css">
#mycontainer
{
    font-size:1em;
    padding:10px;
    width:550px;
    height:150px;
    border:1px solid lightgrey;
    overflow:hidden;
}
#mycontainer h2
{
    margin-top:0px;
    margin-bottom:13px;
    padding:5px;
}
#mycontainer p
{
    margin-top:0px;
    margin-bottom:0px;
}
#scalecont
{
    font-size:0.8em;
}
```

```
#transfercont
{
    display:none;
    font-size:0.8em;
}
#sizecont
{
    display:none;
    font-size:0.8em;
}
.transfer-effect
{
    border:2px solid grey;
}
</style>

</head>
<body>
<div id="mycontainer" class="ui-widget ui-corner-all">
<h2 class="ui-widget-header ui-corner-all">Заголовок</h2>
<p>Съешь еще этих мягких французских булок, да выпей чаю</p>
</div>
<br />
1. Выберите эффект:<br /><br />
<select id="effect1" name="effect">
<option value="scale">Масштаб</option>
<option value="size">Размер</option>
<option value="transfer">Трансфер</option>
</select>
<br /><br />
<div id="scalecont">
1.1. Процент для Scale:
<input id="scalepercent" type="text" value="50"/>
</div>
<div id="transfercont">
1.1. к:
<input id="transl" type="text" value="#but1"/><br /><br />
```

1.2. класс:

```
<input id="trans2" type="text" value="transfer-effect"/>
</div>
```

```
<div id="sizecont">
```

1.1. Ширина:

```
<input id="sizel" type="text" value="200"/>
<br /><br />
```

1.2. Высота:

```
<input id="size2" type="text" value="200"/>
</div>
<br />
```

```
<input id="butl" type="button" value="Применить" />
```

```
<script src="https://code.jquery.com/jquery-3.6.0.min.js">
</script>
```

```
<script src="https://code.jquery.com/ui/1.12.0/jquery-ui.
min.js"></script>
```

```
<script type="text/javascript">
```

```
$(document).ready(function() {
    $("#effect1").change(function() {
        if ($("#effect1").val() === "scale") {
            $("#scalecont").css("display", "block");
            $("#transfercont").css("display", "none");
            $("#sizecont").css("display", "none");
        } else if ($("#effect1").val() === "transfer") {
            $("#transfercont").css("display", "block");
            $("#scalecont").css("display", "none");
            $("#sizecont").css("display", "none");
        } else if ($("#effect1").val() === "size") {
            $("#sizecont").css("display", "block");
            $("#transfercont").css("display", "none");
            $("#scalecont").css("display", "none");
        }
    });
    $("#butl").click(function() {
        var options = {};
        if ($("#effect1").val() === "scale") {
            var options = {percent: 50};
            options.percent = $("#scalepercent").val();
        }
    });
});
```

```
});
```

```
        else if ($("#effect1").val() === "transfer") {
            var options = {to: "#but1", className: "transfer-effect"}
            options.to = $("#trans1").val();
            options.className = $("#trans2").val();
        } else if ($("#effect1").val() === "size") {
            var options = {to: {width: 200, height: 200}};
            options.to.width = $("#size1").val();
            options.to.height = $("#size2").val();
        }
        $("#mycontainer").effect($("#effect1").val(), options
, 1000);
    });

</script>
</body>
</html>
```

12.4. Манипуляция цветом

jQuery UI расширяет возможности jQuery анимации. Теперь Вы можете создавать анимацию, которая может манипулировать цветом.

В анимации теперь могут быть использованы следующие CSS-свойства:

- backgroundColor
- borderColor
- borderBottomColor
- borderLeftColor
- borderRightColor
- borderTopColor
- color
- outlineColor

Никольский А.П., Дубовик Е.В.

Справочник

JAVASCRIPT

КРАТКО # БЫСТРО # ПОД РУКОЙ

Группа подготовки издания:

Зав. редакцией компьютерной литературы: *М. В. Финков*

Редактор: *Е. В. Финков*

Корректор: *А. В. Громова*

12+

ООО «Наука и Техника»

Лицензия №000350 от 23 декабря 1999 года.

192029, г. Санкт-Петербург, пр. Обуховской обороны, д. 107.

Подписано в печать 05.04.2021. Формат 60x90 1/16.

Бумага офсетная. Печать офсетная. Объем 19 п. л.

Тираж 1500. Заказ 739.

Отпечатано ООО «Принт-М»
142300, Московская область,
г. Чехов, ул. Полиграфистов, дом 1

Никольский А.П., Дубовик Е.В.

Справочник JavaScript

КРАТКО # БЫСТРО # ПОД РУКОЙ

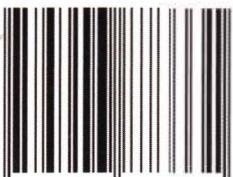
Данный справочник содержит всю ключевую информацию о JavaScript в удобной и наглядной форме. Структура справочника позволяет быстро и удобно находить нужную информацию, получать примеры использования тех или иных элементов и конструкций JavaScript. Отдельное внимание уделено отраслевым рекомендациям по хорошему стилю программирования на JavaScript, объектной модели современных браузеров.

Справочник будет полезен всем, кто использует или изучает JavaScript: от начинающих до профессионалов.

Издательство «Наука и Техника» рекомендует:



ISBN 978-5-94387-333-1



9 78-5-94387-333-1

Издательство "Наука и Техника"
г. Санкт-Петербург

Для заказа книг:
(812) 412-70-26
e-mail: nitmail@nit.com.ru
www.nit.com.ru