JAVASCRIPT ДЛЯ ПОВЕДЕНИЯ

ЧАСТЬ V

В этой части

Глава 21. Введение в JavaScript

Глава 22. Применение JavaScript

ГЛАВА 21

ВВЕДЕНИЕ В JAVASCRIPT

Мэтт Маркус

В этой главе

- Чем является и не является JavaScript
- Переменные и массивы
- Инструкции if/else и циклы
- Собственные и пользовательские функции
- Объекты браузера
- Обработчики событий

В этой главе я познакомлю вас с языком JavaScript. Сейчас вы, возможно, содрогнетесь, и я вас понимаю. Теперь мы на территории полноценного «языка программирования», и это немного пугает. Я обещаю, все не так уж и плохо!

Мы начнем с того, что рассмотрим, чем является и чем не является JavaScript, а также обсудим некоторые способы его использования. Бо́льшая часть главы посвящена введению в синтаксис языка JavaScript — переменным, функциям, операторам, циклам и тому подобному. Начнете ли вы писать код к концу главы? Наверное, нет. Но у вас будет хорошая база, чтобы, увидев сценарий, понять, что в нем происходит. В заключение я рассмотрю некоторые способы управления окном браузера и привязки сценариев к действиям пользователя, таким как щелчок мышью или отправка веб-формы.

Что такое JavaScript?

Если вы продвинулись так далеко в этой книге, то, несомненно, уже знаете, что JavaScript — это язык программирования, добавляющий на сайты интерактивность и пользовательское поведение. Это язык *сценариев, работающих на стороне клиента*, а значит, он запускается на компьютере пользователя, а не на сервере, как другие языки веб-программирования, такие как PHP и Ruby. Это означает, что JavaScript (и то, как мы его используем) зависит от возможностей браузера и настроек. Он может вообще быть недоступен потому, что пользователь предпочел его отключить или потому, что устройство его не поддерживает, хорошие разработчики помнят об этом и учитывают в работе. JavaScript также известен как *динамический* и *слабо типизированный* язык программирования. Не слишком усердно пытайтесь понять это описание, далее я объясню, что оно значит.

Прежде всего, я хочу заявить, что JavaScript не совсем правильно понимают.

Чем не является

Несмотря на свое название, JavaScript не имеет ничего общего с языком программирования Java. Его создал Брэндан Айк, сотрудник ком-

пании Netscape, в 1995 году и первоначально назвал «LiveScript». Однако язык Java был тогда в моде, поэтому ради маркетинга «LiveScript» превратился в «JavaScript». Или просто в «JS», если в разговоре о нем вы хотите выглядеть как можно более профессионально.

Репутация JS не так уж хороша. Некоторое время он был синонимом всевозможных недобросовестных интернет-махинаций — нежелательных переадресаций, раздражающих всплывающих окон, множества туманных «уязвимостей», и это лишь некоторые из них. Было время, когда JavaScript позволял не очень уважающим себя разработчикам проделывать все эти вещи (и даже хуже), но современные браузеры в значительной степени поняли темную сторону развития JavaScript и заблокировали ее. Однако не стоит винить за это сам язык. Как гласит очень старая поговорка: «Кому много дано, с того много и спрашивается». JavaScript всегда предоставлял разработчикам большие возможности по контролю над отображением страниц и поведением браузеров, а насколько ответственно это следует использовать — каждый решает сам.

Чем является

Теперь мы знаем, чем JavaScript не является: он не связан с языком Java, и он вовсе не усатый злодей, прячущийся в вашем браузере, потирая руки в ожидании, когда можно будет отправить сообщение о «горячих девушках вашего города». Давайте поговорим подробнее о том, что же он такое.

JavaScript представляет собой несложный, но невероятно мощный язык сценариев. Чаще всего мы сталкиваемся с ним в браузерах, но JavaScript проник во все, от собственных приложений для PDF-файлов до электронных книг. С помощью JavaScript можно управлять даже веб-серверами.

Как *динамический язык программирования*, JavaScript не нужно запускать через какой-либо компилятор, интерпретирующий понятный человеку код в то, что сможет понять браузер. Последний эффективно считывает код точно так же, как мы, и интерпретирует его на ходу.

Язык JavaScript также *слабо типизированный*. Это означает, что нам не обязательно указывать ему, что такое переменная. Если мы устанавливаем значение переменной равное 5, нам не нужно программно указывать, что эта переменная — число. Как вы, возможно, заметили, 5 — это уже число, и JavaScript распознает его как таковое.

Теперь вам не обязательно запоминать эти термины, чтобы начать писать код на JS, — обратите внимание: я не запоминал. Даже сейчас, когда я их читаю, помню не все. Это нужно всего лишь, чтобы познакомить вас с несколькими терминами, которые вы будете часто слышать, изучая JavaScript, и со временем они начнут обретать для вас все больший смысл.

К тому же будет о чем поговорить на ближайшей вечеринке! «Чем занимаюсь? Ну, я в последнее время всерьез увлекся слабо типизированными динамическими языками сценариев». Люди будут просто молча вам кивать, а это, по-моему, означает, что вы хороший собеседник.

ПРИМЕЧАНИЕ

Язык JavaScript был стандартизирован в 1996 году Европейской ассоциацией производителей компьютеров (European Computer Manufacturer's Association, (ECMA)) и поэтому иногда можно услышать, как его называют *ECMAScript*.

На что способен JavaScript

Чаще всего JavaScript встречается нам как способ добавления интерактивности на страницы. Если наша разметка — «структурный» слой страницы, а ее «презентационный» слой состоит из CSS, третий, «поведенческий», слой составляет JavaScript. Ко всем элементам, атрибутам и тексту на веб-странице можно получить доступ с помощью сценариев, используя объектную модель документа (Document Object Model, DOM), которую мы рассмотрим в главе 22. Мы также сможем написать сценарии, которые будут реагировать на действия пользователя, на ходу изменяя контент страницы, стили CSS или поведение браузера.

Вы наверняка видели их в действии, если когда-либо пытались зарегистрироваться на веб-сайте, вводили имя пользователя и тут же получали ответ, что введенное вами имя уже занято кем-то другим (рис. 21.1). Красная рамка вокруг поля ввода текста и внешний вид сообщения «извините, введенное вами имя уже используется другим пользователем» — примеры JavaScript, изменяющего контент страницы, а блокирование отправки формы — пример JavaScript, изменяющего заданное по умолчанию поведение браузера.

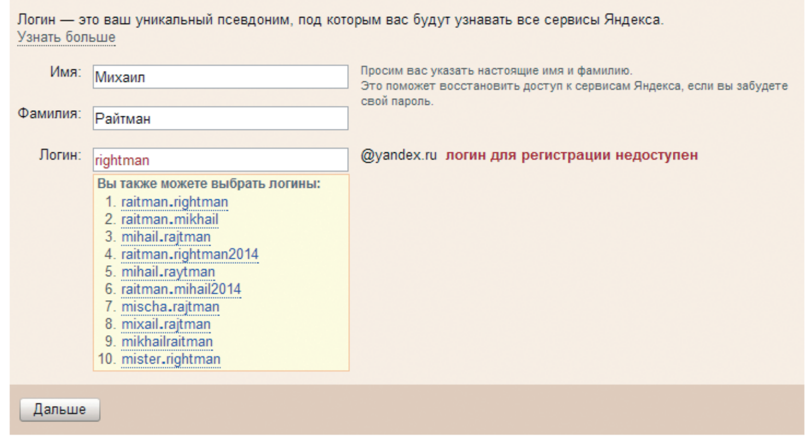

Рис. 21.1. *JavaScript обнаруживает, что регистрируемый адрес электронной почты недоступен, а затем вставляет сообщение и изменяет стили, чтобы сделать проблему очевидной*

Короче говоря, JavaScript позволяет создавать высокоадаптивные интерфейсы, которые улучшают впечатление пользователя и обеспечивают динамическую функциональность, не заставляя ждать, пока сервер загрузит новую страницу. Например, мы можем использовать JavaScript, чтобы выполнить одно из следующих действий:

- Предложить полностью слово, которое пользователь, возможно, вводит в поле поиска, когда он еще его печатает. Это можно увидеть в действии на сайте Google (рис. 21.2).

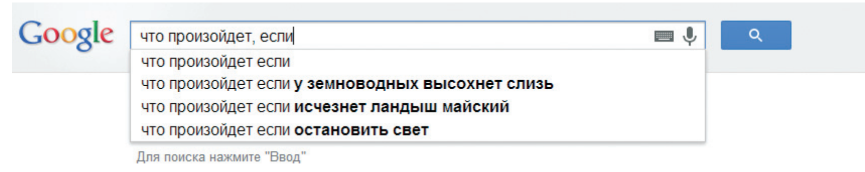

Рис. 21.2. *Сайт Google использует JavaScript, чтобы автоматически предлагать пользователю распространенный поисковый запрос, когда он только начинает его вводить*

- Запрашивать контент и информацию с сервера и вставлять его в текущий документ по мере необходимости, не загружая повторно всю страницу — это обычно называется «Ajax».
- Отображать и скрывать контент после щелчка пользователя мышью по ссылке или заголовку для создания «сворачиваемой» области контента (рис. 21.3).

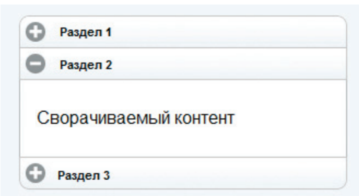

Рис. 21.3. *Сценарии JavaScript можно использовать для отображения и сокрытия части контента*

- Тестировать индивидуальные характеристики и возможности браузеров. Например, можно проверить наличие «события касания», указывающего, что пользователь взаимодействует со страницей через браузер мобильного устройства, и добавить более удобные в плане сенсорного взаимодействия стили и методы.
- Заполнять промежутки там, где не справляются встроенные функции браузера или добавлять в старые браузеры некоторые функции, доступные в новых. Эти виды сценариев обычно называют *прокладками* или *полизаполнениями*.
- Загружать изображение или контент в созданный с помощью пользовательских стилей «лайтбокс» — изолированный на странице с помощью CSS — после того, как пользователь щелкнет мышью по миниатюре изображения (рис. 21.4).

Это далеко не полный список!

Рис. 21.4. *Сценарии JavaScript можно использовать для загрузки изображений в галерею в стиле «лайтбокс»*

Добавление сценариев JavaScript на страницу

Как и CSS, вы можете вставить код сценария прямо в документ или сохранить его во внешнем файле и связать со страницей. В обоих методах используется элемент **script**.

Встроенные сценарии

Чтобы встроить сценарий в страницу, добавьте код как содержимое элемента **script**:

```
<script>
... Здесь будет код JavaScript
</script>
```

Внешние сценарии

Другой метод использует атрибут **src** для указания на файл сценария (с расширением *.js*) по его URL-адресу. В этом случае у элемента **script** нет содержимого.

```
<script src="my_script.js"> </ script>
```

Преимущество внешних сценариев заключается в том, что вы можете применить один и тот же сценарий к нескольким страницам (те же преимущества имеют и внешние таблицы стилей). Недостаток состоит в том, что каждый внешний сценарий требует дополнительного HTTP-запроса с сервера, что замедляет производительность.

Размещение сценариев

Элемент **script** может находиться в любой части документа, но чаще всего он указывается в разделе **head** и в самом конце раздела **body**. Рекомендуется не разбрасывать элементы **script** по всему документу, потому что их будет трудно находить и поддерживать.

Для большинства сценариев предпочтительным является размещение в конце документа, непосредственно перед тегом **</body>**, так как браузер будет осуществлять разбор документа и структуры объектной модели этого документа. Следовательно, эти данные будут готовы и доступны к тому времени, когда браузер дойдет до сценариев и сможет выполнить их быстрее. Кроме того, загрузка и выполнение сценария блокирует визуализацию страницы, так что перемещение его в нижнюю часть улучшает производительность. Тем не менее, в некоторых случаях вам может понадобиться, чтобы сценарий выполнил работу до полной загрузки тела документа, поэтому размещение его в разделе заголовка приведет к лучшей производительности.

Анатомия сценария

Есть веская причина, почему в книге Дэвида Флэнагана «JavaScript. Подробное руководство» (Символ-Плюс, 2008) без малого 1000 страниц. О JavaScript многое можно рассказать! В этом разделе у нас всего лишь несколько страниц для знакомства с основами этого языка, чтобы вы смогли понять сценарии, когда встретите их. Многие разработчики самостоятельно освоили программирование, находя существующие сценарии и адаптируя их под свои нужды.

Немного попрактиковавшись, они готовы начинать писать собственный код с нуля. Вы также можете решить научиться самостоятельно писать код JavaScript в дополнение к навыкам веб-дизайнера. Первый шаг — узнавание фрагментов сценария, так что с этого мы и начнем. Изначально функциональность JavaScript в основном сводилась к грубым методам взаимодействия с пользователем. Для обратной связи,

мы могли бы использовать некоторые встроенные функции JavaScript (рис. 21.5), например **alert()**, чтобы отправить пользователю уведомление и **confirm()**, чтобы попросить его одобрить или отклонить действие. При запросе ввода данных от пользователя наши возможности были более или менее ограничены встроенной функцией **prompt()**. И хотя для этих методов сегодня еще есть время и место, это резкие, навязчивые, и (согласно общему мнению) достаточно неприятные способы взаимодействия с пользователями. Поскольку язык JavaScript со временем развивался, нам оказались доступны гораздо более изящные способы добавления поведения на страницы, создающие у пользователя впечатление более плавной работы.

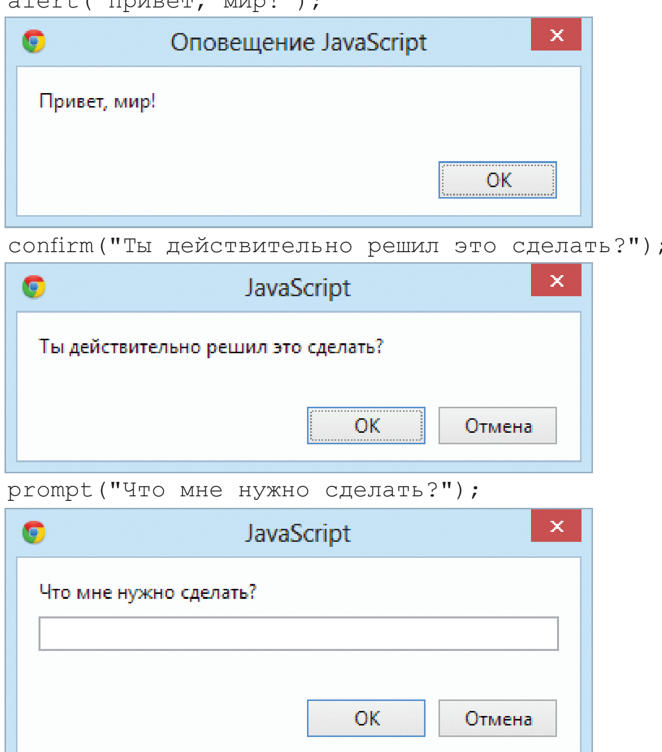

Рис. 21.5. Встроенные функции JavaScript: ***alert()*** *(вверху),* ***confirm()*** *(в центре) и* ***prompt()*** *(внизу)*

Чтобы воспользоваться этими методами взаимодействия, сначала мы должны понять логику сценариев. Это логические модели, общие для всех вариантов языков программирования, хотя синтаксис может варьироваться. Проводя параллель между языками программирования и языками общения — многие грамматические модели являются общими для большинства из них, хотя словарь может меняться от одного языка к другому.

К концу этого раздела вы узнаете о переменных, массивах, операторах сравнения, инструкциях **if/else**, циклах, функциях и многом другом.

ПРИМЕЧАНИЕ

В спецификации HTML 4.01, чтобы быть действительным, тег **script** должен включать в себя атрибут **type**:

```
<script type="text/javascript">…</script>
```

Для XHTML-документов необходимо определить содержимое элемента **script** как CDATA, заключив код в следующую оболочку:

```
<script type="text/javascript">
// <![CDATA[
…Здесь будет код JavaScript
// ]]>
</script>
```

Основы

Существует несколько общих синтаксических правил, которые проведут вас по всему языку JavaScript.

Важно знать, что он *чувствителен к регистру*. Переменные с именем «myVariable», «myvariable» и «MYVariable» будут рассматриваться как три различных объекта. Кроме того, пробельные символы, такие как отступы и пробелы, игнорируются, если они не являются частью строки текста и не заключены в кавычки.

Инструкции

Сценарий состоит из ряда инструкций. Инструкция — это команда, которая сообщает браузеру, что делать. Ниже представлена простая инструкция, указывающая браузеру отобразить уведомление с фразой «Спасибо».

```
alert("Спасибо.");
```

Синтаксис языка JavaScript чувствителен к регистру.

Точка с запятой в конце инструкции сообщает JavaScript, что это конец команды, подобно тому, как точка завершает предложение. Согласно стандарту JavaScript конец строки также приведет к концу команды, но лучшим вариантом считается завершение каждой инструкции точкой с запятой.

Комментарии

Язык JavaScript позволяет оставлять комментарии, которые будут игнорироваться при выполнении сценария, так что вы можете оставить напоминания и разъяснения в своем коде. Это особенно полезно, если в будущем его будет редактировать другой разработчик.

Существует два способа использования комментариев. Для однострочных комментариев используйте два слеша (`//`) в начале строки. Однострочный комментарии можно размещать на одной строке с инструкцией, если он располагается посленее. Его не обязательно закрывать, так как конец строки эффективно это сделает за вас.

```
//Это однострочный комментарий.
```

Многострочные комментарии используют тот же синтаксис, который вам встречался в CSS. Все, что находится между символами `/**/`, игнорируется браузером. Их можно использовать, чтобы «комментировать» заметки и даже фрагменты сценария при поиске ошибок в коде.

```
/* Это многострочный комментарий.

Все, что находится между этими наборами символов, будет полностью игнорироваться при выполнении сценария. Такой вид комментария необходимо закрывать. */
```

Я буду применять обозначения однострочного комментария для добавления кратких пояснений к примерам кода, и мы будем использовать

функцию **alert()**, с которой встречались ранее (рис. 21.5), так, чтобы суметь быстро увидеть результаты нашей работы.

Переменные

Переменная — это контейнер для информации. Вы присваиваете ей имя, а затем значение, которое может быть числом, текстовой строкой, элементом объектной модели документа или функцией — чем угодно, на самом деле. Это дает нам удобный способ в дальнейшем обращаться к переменной по имени. Само значение может быть изменено и переназначено в соответствии с тем, что диктует логика наших сценариев. Следующее определение создает переменную с именем **foo** и присваивает ей значение 5:

```
var foo = 5;
```

Начнем с объявления переменной с помощью ключевого слова **var**. Единственный знак равенства (**=**) означает, что мы присваиваем ей значение. Так как это конец нашего утверждения, мы завершаем строку точкой с запятой. Переменные также можно объявлять без ключевого слова **var**, которое влияет на то, какая часть вашего сценария будет иметь доступ к информации, содержащейся в них. Мы обсудим это дальше в разделе «Область видимости переменной и ключевое слово var» данной главы. В качестве имени переменной можно использовать все, что угодно, но убедитесь, что оно будет иметь смысл для вас в дальнейшем. Не нужно называть переменную **data** или вроде того, имя должно описывать содержащуюся в ней информацию.

В приведенном выше примере имя «numberFive» может оказаться более полезным, чем «foo». Существует несколько правил именования переменных:

Переменная — это контейнер для информации.

- Имя должно начинаться с буквы или символа подчеркивания.
- Оно может состоять из латинских букв, цифр и символов подчеркивания в любом сочетании.
- Оно не может содержать пробелов. В качестве альтернативы вместо пробелов используйте символы подчеркивания или сэкономьте пространство и пишите слова слитно, начиная следующее с заглавной буквы (например, **my_variable** или **myVariable**).
- В имени не должно использоваться специальных символов (**! . , / \ + * =** т.д.).

Значение переменной можно изменить в любой момент, переопределив его в любой позиции сценария. Помните: JavaScript чувствителен к регистру.

Типы данных

Значения, которые мы назначаем переменным, относятся к одному из нескольких различных *типов данных*.

Неопределенный тип данных

Простейший из них, скорее всего, «неопределенный» (**`undefined`**). Если мы определяем переменную, присваивая ей имя, но не значение, она будет содержать значение «undefined».

```
var foo;
alert(foo); // Открывает диалоговое окно, содержащее
"undefined".
```

Скорее всего, прямо сейчас вы найдете немного вариантов применения этого типа данных, но с ним следует познакомиться, чтобы иметь возможность исправить ошибки, которые вы наверняка допустите на первых этапах работы с JavaScript. Если переменная имеет значение «undefined», хотя и не должна, стоит перепроверить, была ли она определена верно и не допущены ли опечатки в имени.

Тип данных NULL

Подобно описанному выше, присвоение переменной значения **`null`** (опять же, с учетом регистра) просто сообщает: «Определите эту переменную, но не присваивайте ей собственного значения».

```
var foo = null;
alert(foo); // Открывает диалоговое окно, содержащее
"null".
```

Числовой тип данных

Переменной можно присвоить числовые значения.

```
var foo = 5;

alert(foo); // Открывает диалоговое окно, содержащее "5".
```

Слово «foo» теперь означает то же самое, что и число пять, применительно к JavaScript. Так как этот язык «слабо типизированный», нам не нужно указывать сценарию воспринимать переменную **`foo`** как число пять.

Переменная ведет себя так же, как число, поэтому с ней можно производить те же действия, что и с любым другим числом, используя классические математические обозначения: **`+`**, **`-`**, **`*`**, и **`/`** для сложения, вычитания, умножения и деления, соответственно. В этом примере мы используем символ плюс (**`+`**), чтобы добавить переменную **`foo`** к самой себе (**`foo`** + **`foo`**).

```
var foo = 5;
alert(foo + foo); // Появится уведомление "10".
```

Строковый тип данных

Еще один тип данных, который можно сохранить в переменной — *строка*, по сути он представляет собой строку текста. Заключение символов в одинарные или двойные кавычки указывает, что это строка, как показано ниже:

```
var foo = "five";
alert( foo ); // Появится уведомление "five"
```

Переменная **foo** теперь воспринимается так же, как слово «five». Это относится к любой комбинации символов: букв, цифр, пробелов и так далее. Если значение заключено в кавычки, оно будет рассматриваться как строка текста. Если заключить число пять (5) в кавычки и присвоить его переменной, она будет вести себя не как число, а как текстовая строка, содержащая символ «5».

Ранее мы видели, что символ плюс (+) используется для сложения чисел. Когда символ плюс используется со строками, он склеивает их вместе (происходит так называетмая *конкатенация*) в одну длинную строку, как показано в этом примере.

```
var foo = "bye"
alert (foo + foo); // Появится уведомление "byebye"
```

Обратите внимание, какое уведомление возвращается в следующем примере, когда мы определяем значение 5 в кавычках, передавая его как строку, а не число.

```
var foo = "5";
alert( foo + foo ); // Появится уведомление "55"
```

Если мы объединим строки и числа, JavaScript решит, что число также следует рассматривать как строку, а значит, произвести математические вычисления будет невозможно.

```
var foo = "five";
var bar = 5;
alert( foo + bar ); // Появится уведомление "five5"
```

Логический тип данных

Также мы можем присвоить переменной значение **true** или **false**. Это *логическое* или *булево значение*, которое является стержнем всей логики. Логические значения используют ключевые слова **true** и **false**, встроенные в JavaScript, поэтому кавычки не нужны .

```
var foo = true; // Переменная "foo" теперь истинна
```

Так же как в случае с числами, если заключить указанное выше значение в кавычки, мы сохраним для нашей переменной *слово* «true», а не собственное значение **true** (то есть не **false**).

В некотором смысле, все в JavaScript имеет собственное значение **true** или **false**. Значения **null**, **undefined**, **0** и пустая строка (**""**) по своей сути ложные, а все остальные значения по сути истинные. Эти значения, хотя и не идентичные логическим понятиям «истина» и «ложь», обычно называют «истинными» и «ложными».

Массивы

Массив представляет собой группу из нескольких значений (называемых *членами*), которые могут быть назначены одной переменной. Значения в массиве называются *индексированными*, то есть к ним можно обращаться по номеру в соответствии с тем, в какой последовательности они появляются в списке. Первому члену присваивается индекс 0,

второму 1 и так далее, поэтому почти всегда окружающие слышат, как программисты начинают считать с нуля — потому что так считает JavaScript и многие другие языки программирования. Помня об этом, можно избежать большой головной боли в будущем. Итак, скажем, нашему сценарию требуются все переменные, определенные ранее. Можно определить их трижды и назвать, например, **foo1, foo2** и так далее или сохранить их в массиве, обозначенном квадратными скобками ([]).

```
var foo = [5, "five", "5"];
```

Теперь, как только вам потребуется получить доступ к любому из этих значений, их можно извлечь из единого массива **foo**, указав их индекс:

```
alert( foo[0] ); // Появится уведомление  "5"
alert( foo[1] ); // Появится уведомление  "five"
alert( foo[2] ); // Появится уведомление  "5"
```

Операторы сравнения

Теперь, когда мы знаем, как сохранить значения в переменных и массивах, следующий логический шаг — научиться сравнивать эти значения. Существует набор специальных символов, называемых *операторами сравнения*, которые позволяют оценить и сравнить значения различными способами:

==	Равно
!=	Не равно
===	Идентично (равно и относится к тому же типу данных)
!==	Не идентично
>	Больше, чем
>=	Больше или равно
<	Меньше, чем
<=	Меньше или равно

Существует причина, почему все эти определения читаются как часть инструкции. При сравнении значений мы делаем предположение, цель которого — получить результат, являющийся, собственно, истинным или ложным. Когда мы сравниваем два значения, JavaScript оценивает утверждение и возвращает нам булево значение, в зависимости от того, является ли утверждение истинным или ложным.

```
alert( 5 == 5 ); // Появится уведомление  "true"
alert( 5 != 6 ); // Появится уведомление  "true"
alert( 5 < 1 ); // Появится уведомление  "false"
```

Равно или идентично

Хитрость в том, чтобы понять разницу между «равно» (==) и «идентично» (===). Мы уже узнали, что все эти значения относятся к определенному типу данных. Например, строка "5" и число 5 схожи, но это не одно и то же.

И именно это нужно выяснить с помощью оператора ===.

```
alert( "5" == 5 ); // Появится уведомление  "true".
Оба значения "5".

alert( "5" === 5 ); // Появится уведомление  "false".
Оба значения "5", но они относятся к разным типам данных.

alert( "5" !== 5 ); // Ответ будет "true", так как они от-
носятся к разным типам данных.
```

Даже если вам придется прочитать это несколько раз, понимание предыдущего предложения будет означать, что вы уже понемногу становитесь программистом.

Математические операторы

Другой тип оператора — *математический*, который выполняет математические функции с числовыми значениями. Мы вкратце касались простых математических операторов сложения (+), вычитания (-), умножения (*) и деления (/). Кроме того, существуют некоторые полезные сочетания, о которых следует знать:

ПРЕДУПРЕЖДЕНИЕ

Будьте осторожны, чтобы случайно не использовать одинарный знак равенства, иначе вы переопределите значение первой переменной для второй.

`+=`	Прибавляет значение к самому себе
`++`	Увеличивает значение числа (или переменной, содержащей числовое значение) на 1.
`--`	Уменьшает значение числа (или переменной, содержащей числовое значение) на 1

Инструкции `if/else`

Инструкции ***`if/else`*** — это способ заставить JavaScript задать себе вопрос на проверку истинности. В той или иной степени они — основа всей современной логики, которая может быть записана в JavaScript, и они настолько просты, насколько это возможно в программировании. Фактически, они написаны на простом английском языке. Структура условной инструкции выглядит следующим образом.

```
if( true ) {
// Сделайте что-нибудь.
}
```

Идиоматический JavaScript

В сообществе пользователей JavaScript предпринимаются усилия по созданию пособия по стилям для написания кода JavaScript. В документе «Принципы написания согласованного, идиоматического JavaScript» говорится следующее: "Весь код программы должен выглядеть так, будто он набран одним человеком, независимо от того, сколько людей участвовали в ее разработке». Для достижения этой цели группа разработчиков составила Манифест идиоматического стиля, в котором описывается, как следует прописывать пробельные символы, переводы строк, кавычки, функции, переменные и прочее для создания «прекрасного кода». Узнайте об этом больше на сайте **github.com/rwldrn/idiomatic.js/**.

Инструкция сообщает браузеру: «Если данное условие удовлетворено, следует выполнить команды, перечисленные между фигурными скобками ({})». Помните, что в JavaScript не важны пробельные символы в коде, поэтому пробелы с обеих сторон **`( true )`** добавлены только для того, чтобы сделать код более удобочитаемым.

Ниже приведен простой пример использования указанного ранее массива:

```
var foo = [5, "five", "5"];

if( foo[1] === "five" ) {

alert("Это слово five, написанное обычными буквами");

}
```

Поскольку мы сравниваем, JavaScript отобразит значение **`true`** или **`false`**. Выделенная строка кода сообщает, что «истина или ложь: идентична ли переменная **`foo`** с индексом **`1`** слову «five»?».

В этом случае уведомление появится потому, что переменная **`foo`** с индексом **`1`** (вторая в списке, если помните) идентична слову «five». В данном случае переменная действительно истинна и появится уведомление.

Мы также можем четко проверить истинность чего-либо, используя оператор сравнения **`!=`**, означающий «не равно».

```
if( 1 != 2 ) {

alert("Если вы этого не видите, у нас проблемы серьезнее, чем JavaScript.");

// 1 никогда не равно 2, поэтому данное уведомление  должно появляться всегда.

}
```

Javascript сообщает: «инструкция «1 не равно 2» истинна, поэтому я запускаю данный код».

Если утверждение не расценивается как «**`true`**» код внутри фигурных скобок будет пропущен полностью:

```
if( 1 == 2 ) {

alert("Если вы этого не видите, у нас проблемы серьезнее, чем JavaScript.");

// 1 не равно 2, поэтому данный код никогда не запустится.

}
```

Суть инструкции `if` ясна, а что насчет `else`?

Как поступить, если нам нужно сделать что-то одно, если результат истинный, и что-то другое, если ложный? Можно написать две инструкции, но это немного неудобно. Вместо этого лучше просто сказать «если нет, сделай что-нибудь... другое».

```
var test = "testing";
if( test == "testing" ) {
alert( "Вы ничего не поменяли" );
} else {
alert( "Вы что-то поменяли!" );
}
```

Изменение значения переменной **testing** на другое, кроме слова «testing» вызовет уведомление «Вы что-то поменяли!».

Циклы

Бывают случаи, когда необходимо пройтись по всем элементам в массиве и с каждым совершить действие, но не хочется выписывать все элементы в список и повторяться десять раз и более. Далее вы изучите технику, обладающую разрушительной силой: *циклы*.

Возможно, благодаря мне циклы выглядят более захватывающими, чем они кажутся, но они невероятно полезны. Благодаря уже изученному материалу у нас уже неплохо получается работать с отдельными переменными, но дальше мы не продвинемся. Циклы позволяют легко работать с огромными наборами данных.

Скажем, у нас есть форма, в которой ни одно из полей не должно оставаться пустым. Если мы используем DOM для всех случаев ввода текста на странице, она предоставит массив для каждого элемента ввода текста. (Я скажу вам больше о том, как сделать это с помощью DOM в следующей главе.) Конечно, можно проверять каждое значение, сохраненное в этом массиве поочередно, но тогда код будет очень длинным, а его поддержка превратится в кошмар. Если мы используем цикл для проверки каждого значения, нам не придется изменять сценарий, независимо от того, сколько полей добавлены на страницу или удалены с нее. Циклы позволяют применить действие к каждому элементу в массиве, независимо от размера этого массива. Существует несколько способов написания циклов, но одним из наиболее популярных является метод **for**. Основная структура цикла **for** выглядит следующим образом:

```
for( инициализации переменной; проверки условия; изменения значения; )
{
// выполнить какое-либо действие
}
```

Ниже приведен пример цикла **for** в действии.

```
for( var i = 0; i <= 2; i++ ) {
alert( i ); // Этот цикл вызовет три уведомления, с сообщением "0", "1" и "2" соответственно.
}
```

УПРАЖНЕНИЕ 21.1 | АЗЫ РАБОТЫ С JAVASCRIPT

В этом коротком упражнении вы сможете лучше понять переменные, массивы и инструкцию **`if/else`**, переведя выражения, написанные на английском, в код JavaScript. Ответ вы найдете в конце главы.

1. Создайте переменную с именем **`friends`** и назначьте ей массив с именами четырех ваших друзей.
2. Отобразите диалоговое окно, в котором указано третье имя из списка **`friends`**.
3. Создайте переменную **`name`** и присвоите ей строковое значение, содержащее ваше имя.
4. Если значение **`name`** идентично «Алиса», выведите диалоговое окно, сообщающее «Меня тоже так зовут!»
5. Создайте переменную **`myVariable`** и присвойте ей числовое значение в диапазоне между 1 и 10. Если значение переменной **`myVariable`** больше пяти, отобразите диалоговое окно с уведомлением «больше». Если нет, покажите диалоговое окно «меньше».

Материала довольно много, поэтому давайте разберем его по частям.

`for ()`

Во-первых, мы вызываем инструкцию **`for ()`**, встроенную в JavaScript. Она сообщает: «Каждый раз, когда это верно, сделайте так». Далее мы должны добавить к инструкции немного данных.

`var i = 0;`

Создает новую переменную, **`i`**, со значением равным нулю. Вы можете определить, что это переменная, по одному знаку равенства. Чаще всего вам будет встречаться использование программистами буквы «i» (сокращения от «индекс») в качестве имени переменной, но имейте в виду, что вы можете использовать вместо него любое. Это распространенное условное обозначение, но не правило.

Зададим начальное значение «0» потому, что мы хотим по-прежнему начинать отсчет с нуля. В конце концов, так считает JavaScript.

`1 <= 2;`

Выражением **`i <= 2;`**, мы говорим, что «до тех пор, пока **`i`** меньше или равно 2, нужно продолжать цикл». Поскольку мы начинаем считать с нуля, это означает, что цикл будет выполняться трижды.

`i++`

Наконец, **`i++`** — это сокращение, означающее «каждый раз, когда запускается этот цикл, добавьте одно из значений **`i`** (++ — это одно из сочетаний математических операторов, с которым мы встречались раньше). Без этого шага, **`i`** всегда равно нулю, и цикл будет длиться вечно! К счастью, современные браузеры достаточно умны, чтобы не позволить этому случиться. Если одна из этих трех частей отсутствует, цикл совсем не будет выполняться.

`{`*`сценарий`*`}`

Все, что находится внутри этих фигурных скобок, выполняется один раз для каждого повтора цикла, которых в данном случае три. Эта переменная **`i`** также доступна для использования в коде, который выполняется циклом, как мы увидим дальше. Давайте вернемся к примеру «проверить каждый элемент массива». Как написать цикл, который сделает это за нас?

```
var items = ["foo", "bar", "baz"]; // Сначала создадим массив.
for( var i = 0; i <= items.length; i++ ) {
alert( items[i] ); // Создаст уведомление для каждого элемента в массиве.
}
```

Этот пример отличается от нашего первого цикла двумя ключевыми моментами:

`items.length`

Вместо использования числа для ограничения количества запусков цикла мы используем свойство, встроенное в JavaScript, для определения «длины» нашего массива, то есть количества содержащихся в нем элементов. **`.length`** — всего лишь одно из стандартных свойств и методов объекта **`Array`** в JavaScript.

`items[i]`

Помните, я говорил, что мы можем использовать переменную **`i`** внутри цикла? Ну, мы можем использовать ее, чтобы сослаться на каждый индекс в массиве. Хорошо, что мы начали отсчет с нуля, если бы было установлено начальное значение **`i`** равное 1, первый элемент в массиве оказался бы пропущен.

Теперь, независимо от того, насколько большим или маленьким должен стать массив, цикл станет выполняться ровно столько раз, сколько элементов в массиве, и всегда будет содержать удобную ссылку на каждый из них.

Существуют буквально десятки способов написания цикла, но это — один из наиболее распространенных методов, который встретится вам во Всемирной паутине. Разработчики используют циклы для выполнения ряда задач, таких как:

- Запуск цикла по списку элементов на странице и проверка значения каждого из них путем применения стиля или добавления/удаления/изменения атрибутов. Например, можно запустить цикл по всем элементам веб-формы и убедиться, что пользователи ввели допустимое значение для каждого, прежде чем они смогут перейти дальше.
- Создание в исходном массиве нового массива элементов, которые имеют определенные значения. Мы проверяем значение каждого элемента исходного массива в пределах цикла и, если значение соответствует тому, которое мы ищем, заполняем новый массив только этими элементами, что превращает цикл в своего рода фильтр.

Функции

Я уже незаметно познакомил вас с несколькими функциями. Ниже приведен пример функции, которую вы, возможно, узнаете:

```
alert ("Я была функцией все это время!");
```

Функция — это фрагмент кода, который не работает, пока к нему не обратятся или не вызовут. Функция **`alert()`** встроена в наш браузер. Это блок кода, который выполняется, только когда мы четко скажем ему действовать. В некотором смысле можно представить функцию как переменную, которая содержит *логику*, заключающуюся в том, что ссылка на эту переменную будет запускать весь код, хранящийся внутри нее.

Для всех функций характерен общий шаблон (рис. 21.6). Имя функции всегда сопровождается скобками (без пробела), а затем — парой

фигурных скобок, содержащих относящийся к функции код. В скобках иногда указана дополнительная информация, используемая функцией и называемая *аргументами*.

Аргументы — это данные, которые могут влиять на поведение функции. Например, функция **alert** в качестве аргумента принимает строку текста, и использует эту информацию для заполнения итогового диалогового окна.

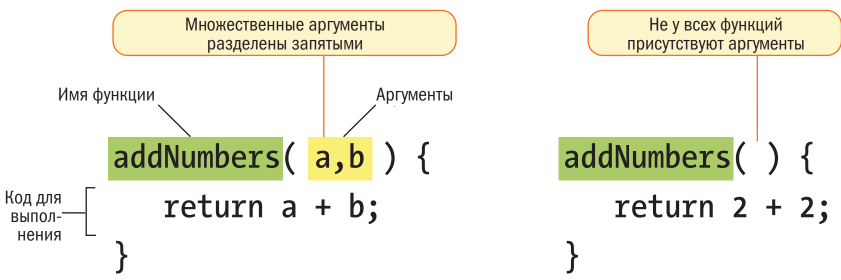

Рис. 21.6. *Структура функции*

Существует два типа функций: «готовые» (собственные функции JavaScript), и те, которые вы создаете самостоятельно (пользовательские функции). Давайте рассмотрим оба.

Собственные функции

В JavaScript существуют сотни встроенных предопределенных функций, в том числе:

alert(), **confirm()** и **prompt()**

Эти функции запускают диалоговые окна на уровне браузера.

Date()

Возвращает текущую дату и время.

parseInt("123")

Эта функция, среди прочего, позволяет принимать строковый тип данных, содержащий числа, и превращает его в числовой тип данных. Строка передается в функцию как аргумент.

setTimeout(*имяФункции*, 5000)

Будет выполнять функцию с задержкой. Функция указывается в первом аргументе, а задержка задается в миллисекундах во втором (в примере 5000 миллисекунд равно 5 секундам).

Кроме того, существует множество других функций.

Пользовательские функции

Чтобы создать пользовательскую функцию, введите ключевое слово **function**, укажите ее имя, добавьте открывающую и закрывающую скобки, а потом — открывающую и закрывающую фигурные скобки.

```
function name() {
// Здесь будет наш код функции.
}
```

Как и в случаях с переменными и массивами, имя функции может быть любым, но к нему применимы все те же правила синтаксиса.

Если бы нам пришлось создавать функцию, просто отображающую уведомление с определенным текстом (которых, я знаю, и так слишком много) она бы выглядела так:

```
function foo() {
alert("Наша функция только что была запущена!");
// Код не запустится, пока мы не вызовем функцию 'foo()'
}
```

Далее мы сможем вызывать эту функцию и выполнять указанный в ней код в любой позиции нашего сценария, написав следующее:

```
foo(); // Уведомление " Наша функция только что была запу-
щена!"
```

Мы можем вызывать функцию в коде любое количество раз. Это значительно экономит время и избавляет от избыточного кодирования.

Аргументы

Наличие функции, выполняющей один и тот же код на протяжении всего сценария, скорее всего, будет вовсе не так полезно. Мы можем «передавать аргументы» (обеспечивать данные) собственным и пользовательским функциям для того, чтобы применить логику функции к различным наборам данных в разное время.

Чтобы оставить место для аргументов, добавьте в скобки одну или несколько переменных, разделенных точкой с запятой, при определении функции. Потом, когда мы вызовем ее, все, что заключено в скобках, будет передано этой переменной при выполнении функции. Это может звучать немного непонятно, но все не так сложно, когда вы видите функцию в действии.

Аргумент — это значение или данные, которые используются при выполнении функции.

Например, предположим, нам необходимо создать очень простую функцию, которая создает уведомления для определенного количества элементов, находящихся в массиве. Мы уже узнали, что для получения числа элементов в массиве можно использовать **.length**, поэтому нам просто нужно найти способ, как измерить массив с помощью функции. Это делается путем предоставления массива для измерения в качестве аргумента. Для того при определении пользовательской функции мы определяем в скобках имя переменной. Затем она станет доступна в функции и будет содержать любой аргумент, который мы передадим при вызове последней.

```
function alertArraySize(arr) {
alert(arr.length);
}
```

Теперь любой массив, указанный в скобках, когда мы вызываем функцию, будет передаваться в нее с именем переменной **arr**. Нам осталось только определить его длину.

```
var test = [1,2,3,4,5];
alertArraySize(test); // Уведомление "5"
```

Возвращение значения

Эта часть особенно непонятна, но невероятно полезна. Довольно часто функцию используют для вычисления, а затем получают значения, которые можно применить в другом месте в сценарии. Это можно сделать, используя уже имеющиеся знания и грамотно применив переменные, но есть более простой способ.

Ключевое слово **return** внутри функции эффективно превращает ее в переменную с динамической ценностью! Это проще показать, чем рассказать, так что оставайтесь со мной, пока мы изучаем следующий пример.

```
function addNumbers(a,b) {
return a + b;
}
```

Теперь у нас есть функция, которая принимает два аргумента и складывает их. Если результат останется внутри функции, пользы от него не будет, потому что мы не сможем использовать этот результат больше нигде в сценарии. В данном случае мы применяем ключевое слово **return**, чтобы вывести результат из функции. Сейчас при любой ссылке на нее вы получите результат функции — как если бы это была переменная.

```
alert( addNumbers(2,5) ); // Уведомление "7"
```

В некотором смысле, функция **addNumbers** теперь переменная, содержащая динамическое значение — значение нашего вычисления. Если мы не вернем его внутрь функции, относительно предыдущего сценария появится уведомление «undefined», как и в случае с переменной, которой не задано значение.

Ключевое слово **return** с подвохом. Как только JavaScript видит, что пришло время вернуть значение, функция заканчивается. Рассмотрим следующий пример:

```
function bar() {
return 3;
alert("Мы никогда не увидим это уведомление ");
}
```

При вызове этой функции с использованием **bar()**, уведомление во второй строке никогда не сработает. Функция закончится, как только JavaScript решит, что пора возвращать значение.

Область видимости переменной и ключевое слово var

Иногда необходимо, чтобы переменная, которую вы определили в функции, была доступна по всему сценарию. В других случаях вам может понадобиться ограничить ее и сделать доступной только для функции, в которой она находится. Понятие доступности переменной известно как *область видимости*. Переменная, которую могут использовать все сценарии на странице, является *глобальной*, а переменная, доступная только в пределах своей родительской функции — *локальной*. Переменные JavaScript используют функции для управления областью видимости. Если переменная определена вне функции, она будет глобальной и доступной для всех сценариев. Когда вы определяете переменную внутри функции и хотите, чтобы она использовалась только ею, можно пометить эту переменную как локальную, добавив ключевое слово **var** перед ее именем.

```
var foo = "значение";
```

Чтобы превратить переменную, указанную в функции, в глобальную, мы опускаем ключевое слово **var** и просто определяем переменную:

```
var foo = "значение";
```

Следует с осторожностью подходить к тому, как вы определяете переменные внутри функции, иначе это может привести к неожиданным результатам. Например, возьмем следующий фрагмент кода JavaScript:

```
function double (num) {
total = num + num;
return total;
}
var total = 10;
var number = double (20);
alert (total); // Уведомление 40.
```

Можно ожидать, что, поскольку вы специально присвоили переменной **total** значение 10, функция **alert(total)** в конце сценария отобразится число 10. Но поскольку мы не ограничили область видимости переменной **total** в функции с помощью ключевого слова **var,** ее область видимости оказывается глобальной. Поэтому несмотря на то, что переменной **total** задано значение 10, следующий за ней оператор запускает функцию и захватывает значение переменной **total**, указанной в ней. Без ключевого слова **var** переменная «просачивается» в глобальную область видимости.

Как уже стало понятно, проблема с глобальными переменными в том, что они будут общими для всех сценариев на странице. Чем больше переменных, просачивающихся в глобальную область видимости, тем больше шансов, что произойдет «столкновение», в котором переменная,

Сокрытие переменной из глобальной области видимости

Если вы хотите быть уверены, что ни одна из ваших переменных не окажется глобальной, заключите весь свой код JavaScript в следующую оболочку:

```
<script>
(function() {
//Весь код здесь!
})();
<script>
```

Такой вот небольшой карантин называется *IIFE* (Independently Invoked Functional Expression) —этот метод и связанный с ним термин появился благодаря Бену Алману.

указанная в другом месте (даже в другом сценарии) совпадет с одной из ваших. Это может привести к тому, что переменные будут случайно переопределены с неожиданными значениями, которые станут причиной ошибок в сценарии.

Помните, что не всегда возможно контролировать весь используемый код. Часто страницы содержат код, написанный третьими лицами, например:

- Сценарии для визуализации рекламы
- Сценарии отслеживания пользователей и анализа
- Кнопки «поделиться» социальных медиа.

Рекомендуется не доводить до столкновения переменных, поэтому, когда вы начинаете писать собственные сценарии, по возможности оставляйте свои переменные локальными (см. врезку).

На этом мы завершаем наш краткий (ну, хорошо, не такой уж краткий) вводный тур по синтаксису языка JavaScript. На самом деле многое не затронуто, но этот материал должен дать вам достойную основу для самостоятельного получения дополнительной информации и возможности истолковывать сценарии, когда вы с ними встретитесь. Нам нужно освоить еще несколько функций, связанных с JavaScript, прежде чем рассматривать примеры.

Объект браузера

Язык JavaScript не только способен контролировать элементы на веб-странице, он также предоставляет вам доступ к самому окну браузера и возможность манипулировать его частью. Например вы можете получить или заменить URL-адрес, указанный в адресной строке браузера. В JavaScript браузер известен как объект **`window`**. Окно браузера имеет ряд свойств и методов, которые мы можем использовать для взаимодействия с ним. В самом деле, фактически функция **`alert()`** является одним из стандартных методов объектов браузера. В табл. 21.1 перечислены лишь некоторые из свойств и методов, которые можно использовать с объектом **`window`**, чтобы дать вам представление о его возможностях.

Табл. 21.1. *Свойства и методы браузера*

Свойство/Метод	**Описание**
`event`	Представляет состояние события
`history`	Содержит URL-адреса, посещенные пользователем в окне браузера
`location`	Предоставляет доступ для чтения/записи URL-адреса в адресной строке

Свойство/Метод	Описание
`status`	Задает или возвращает текст в строке состояния окна браузера
`alert()`	Отображает уведомление с заданным сообщением и кнопку **OK**
`close()`	Закрывает текущее окно
`confirm()`	Отображает диалоговое окно с определенным сообщением, а также кнопки **OK** и **Отмена**
`focus()`	Устанавливает фокус на текущее окно

События

JavaScript имеет доступ к объектам на странице и в окне браузера, но знаете ли вы, что он также «прислушивается», когда происходят определенные события? *Событие* — это действие, которое можно обнаружить с помощью JavaScript, например когда загружается документ или пользователь щелкает мышью по элементу или просто наводит на него курсор. Язык HTML 4.0 позволил привязать сценарий к событиям на странице, будь то события, инициированные пользователем, самим браузером или другие сценарии. Это называется *привязка событий*.

Обработчики событий «прислушиваются» к определенным действиям документа, браузера или пользователя и привязывают к этим действиям сценарий.

В сценариях события определяются обработчиками событий. Например обработчик событий **`onload`** запускает сценарий при загрузке документа, а обработчики событий **`onclick`** и **`onmouseove`** запускают сценарий, когда пользователь щелкает мышью по элементу или наводит на него курсор, соответственно. В табл. 21.2 перечислены некоторые из наиболее распространенных обработчиков событий.

Табл. 21.2. *Распространенные события*

Свойство/Метод	Описание
`onblur`	Элемент потерял фокус
`onchange`	Меняется содержимое поля веб-формы
`onclick`	Осуществлен щелчок мышью по объекту
`onerror`	При загрузке документа или изображения возникает ошибка
`onfocus`	Элемент оказывается в фокусе
`onkeydown`	Нажата клавиша на клавиатуре
`onkeypress`	Клавиша на клавиатуре нажата или удерживается
`onkeyup`	Клавиша на клавиатуре отпущена
`onload`	Завершена загрузка страницы или изображения

Свойство/Метод	Описание
onmousedown	Нажата кнопка мыши
onmousemove	Указатель мыши перемещен
onmouseout	Указатель мыши смещен с элемента
onmouseover	Указатель мыши наведен на элемент
onmouseup	Кнопка мыши отпущена.
onsubmit	Нажата кнопка отправки данных веб-формы

Существует три распространенных метода применения обработчиков событий к элементам на страницах:

- В качестве атрибутов HTML;
- В качестве метода, связанного с элементом;
- С помощью метода **addEventListener**.

В примерах двух последних подходов используется объект **window**. Любые события, присоединенные к **window**, применяются ко всему документу. Во всех этих случаях мы также используем событие **onclick**.

Как атрибут HTML

Функцию, которую необходимо запустить, можно указать в атрибуте в разметке, как показано в следующем примере.

```
<body onclick="myFunction();"> /* Функция myFunction будет запускаться, когда пользователь щелкнет мышью по любому элементу в разделе body */
```

Этот способ присоединения событий к элементам на странице, хоть и функционален, но уже устарел. Его следует избегать по той же причине, по которой мы стараемся не добавлять в разметку атрибуты **style** для применения стилей к отдельным элементам. В этом случае он стирает грань между семантическим и поведенческим слоями наших страниц и может быстро превратить процесс техподдержки в кошмар.

Как метод

Еще один несколько устаревший подход к присоединению событий, хотя он строго удерживает элементы в пределах сценариев. Кроме того, функции можно присоединить, используя помощники, уже встроенные в JavaScript.

```
window.onclick = myFunction; /* Функция myFunction будет запускаться, когда пользователь щелкнет мышью по любому элементу в окне браузера */
```

Также можно использовать анонимную функцию вместо предопределенной

```
window.onclick = function() {

/* Любой размещенный здесь код будет запускаться когда поль-
зователь щелкнет мышью по любому элементу в окне браузера */

};
```

У такого подхода есть преимущество простоты и удобства технической поддержки, но имеется и довольно существенный недостаток: с помощью него можно привязывать только по одному событию за один раз.

```
window.onclick = myFunction;

window.onclick = myOtherFunction;
```

В приведенном выше примере второе привязанное событие отменяет первое, поэтому, когда пользователь щелкнет мышью в окне браузера, запустится только функция **myOtherFunction**. Ссылка на функцию **myFunction** отбрасывается.

AddEventListener

На первый взгляд, этот подход кажется более сложным, однако, он позволяет нам сохранить логику сценариев и выполнить несколько привязок к одному объекту. Синтаксис чуть более подробный. Мы начинаем с вызова метода **addEventListener** целевого объекта, а затем указываем интересующее нас событие и выполняемую функцию в виде двух аргументов.

Для получения дополнительной информации о методе **addEventListener** посетите веб-страницу **developer.mozilla.org/en/DOM/element.addEventListener**.

```
window.addEventListener("click", myFunction);
```

Обратите внимание, что в этом синтаксисе мы опускаем предлог «on» в имени обработчика событий. Как и в предыдущем методе, **AddEventListener** можно использовать также и с анонимными функциями,

```
window.addEventListener("click", function(e) {

});
```

Резюме

Теперь вы познакомились со многими важными основными понятиями языка JavaScript. Вы увидели переменные, типы данных и массивы, повстречались с инструкциями **if/else**, циклами и функциями и сможете отличить объекты браузера от обработчиков событий. Давайте рассмотрим несколько примеров простых сценариев, чтобы увидеть, как они объединяются вместе.

Пример 1: Повесть о двух аргументах

Ниже приведена простая функция, которая принимает два аргумента и возвращает большее из двух значений.

```
greatestOfTwo( first, second ) {
if( first > second ) {
return first;
} else {
return second;
}
}
```

Начнем с присвоения нашей функции имени: «greatestOfTwo». Мы установим, что она должна принимать два аргумента, которые называются просто «first» и «second» за неимением более подходящих описательных слов. Функция содержит инструкцию **`if/else`**, который возвращает значение «first», если первый аргумент больше, чем второй, и «second» — если это не так.

Пример 2: Самое длинное слово

Ниже приведена функция, принимающая массив строк как один аргумент и возвращающая самую длинную из них.

```
longestWord( strings ) {
var longest = strings[0];
for( i = 1; i < strings.length; i++ ) {
if ( strings[i].length > longest.length ) {
longest = strings[i];
}
}
return longest;
}
```

Сначала мы именуем функцию и позволяем ей принять один аргумент. Затем задаем переменной **`longest`** исходное значение первого элемента массива: **`strings[0]`**. Мы начинаем цикл с 1 вместо 0, так как первое значение массива уже используется. При каждой итерации цикла мы сравниваем длину текущего элемента массива с длиной значения, сохраненного в переменной **`longest`**. Если текущий элемент в массиве содержит больше символов, чем текущее значение переменной **`longest`**, мы заменяем значение **`longest`** этим элементом. Если нет, то мы ничего не делаем. После завершения цикла возвращается значение переменной **`longest`**, которое теперь будет содержать самую длинную строку в массиве.

УПРАЖНЕНИЕ 21.2. ДЛЯ САМОСТОЯТЕЛЬНОГО ВЫПОЛНЕНИЯ

В этом упражнении вы напишете сценарий, который будет обновлять заголовок страницы в окне браузера с помощью счетчика новых сообщений. Вероятно, время от времени вы сталкиваетесь с подобными сценариями на реальных сайтах. Ради успеха упражнения будем считать, что однажды этот сценарий станет частью более крупного веб-приложения, а нам поручено только обновлять заголовок страницы текущей информацией о числе непрочитанных сообщений.

Я уже создала для вас документ (*title.html*), который доступен в папке материалов к данной главе на диске, прилагающемся к книге.

1. Сначала откройте файл *title.html* в браузере. Вы увидите, пустую страницу с уже заполненным тегом **title**. Надпись в верхней части окна браузера гласит: «Веб-приложение».
2. Теперь откройте документ в текстовом редакторе. Прямо перед закрывающим тегом **</body>** вы увидите элемент **script**, содержащий комментарий, который вы можете смело удалить.
3. Если мы собираемся менять заголовок страницы, сначала необходимо сохранить оригинал. Для этого следует создать переменную с именем **originalTitle**. В качестве ее значения браузер использует название документа, полученное с помощью метода DOM **document.title**. Теперь у нас есть сохраненная ссылка на заголовок страницы на момент ее загрузки. Эта переменная должна быть глобальной, поэтому мы объявим ее вне всех функций.

```
var originalTitle = document.title;
```

4. Далее следует определить функцию, чтобы иметь возможность повторно использовать сценарий при необходимости. Присвойте ей легко запоминающееся имя, чтобы с первого взгляда понимать, что делает эта функция, когда она вам встретится в коде позднее. Мне нравится имя **showUnreadCount**, но вы можете назвать ее как хотите.

```
var originalTitle = document.title;
function showUnreadCount() {
}
```

5. Нужно подумать, что необходимо, чтобы сделать функцию полезной. Эта функция совершает какое-то действие со счетчиком непрочитанных сообщений, поэтому ее аргументом является одно число, обозначенное **unread** в данном примере.

```
var originalTitle = document.title;
function showUnreadCount( unread ) {
}
```

6. Теперь добавим код, запускающий данную функцию. Нам нужен заголовок документа, чтобы отобразить название документа на странице, плюс счетчик непрочитанных сообщений. Похоже, это работа для конкатенации (+)! Здесь следует задать, что значение **document.title** является (=) любой строкой, сохраненной в переменной **originalTitle** плюс число в функции **showUnreadCount**. Как мы узнали ранее, JavaScript объединяет строку и число, как если бы оба эти значения были строками.

```
var originalTitle = document.title;
function showUnreadCount( unread ) {
document.title = originalTitle + unread;
}
```

7. Давайте опробуем наш сценарий, прежде чем двигаться дальше. Ниже под кодом, где вы определили функцию и переменную **originalTitle**, введите **showUnreadCount(5);**. Теперь сохраните страницу и загрузите ее в браузере (рис. 21.7).

```
var originalTitle = document.title;
function showUnreadCount( unread ) {
document.title = originalTitle + unread;
}
showUnreadCount(5);
```

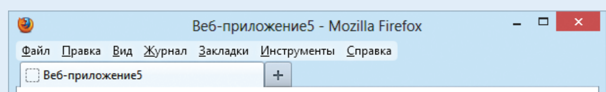

Рис. 21.7. Наш заголовок изменился! Хотя он еще не совсем такой, как нужно

8. Сценарий работает, но его нелегко прочитать. К счастью, нет никаких ограничений на количество строк, которые можно объединить. Здесь мы добавим дополнительные строки, заключим значение счетчика и слова «новых сообщений» в скобки (рис. 21.8).

```
var originalTitle = document.title;
function showUnreadCount( unread ) {
document.title = originalTitle + " (" +
unread
+ " новых сообщений!) ";
}
showUnreadCount(5);
```

Рис. 21.8. Результат

Для дополнительного чтения

Уже представляете, что сотворите с помощью JavaScript? Во Всемирной паутине определенно нет недостатка в руководствах по этому языку, которые помогут вам начать. Я также рекомендую следующие книги:

- «JavaScript. Подробное руководство» Дэвида Флэнагана (Символ-Плюс, 2013);
- «JavaScript и jQuery. Исчерпывающее руководство» Дэвида Сойера Макфарланда (Эксмо, 2012);
- «JavaScript: сильные стороны» Дугласа Крокфорда (Питер, 2013).

Ответы к упражнениям

Упражнение 21.1

1. `var friends = ["name", "othername", "thirdname", "lastname"];`
2. `alert(friends[2]);`
3. `var name = "yourName";`
4. `if( name === Алиса) { alert("Меня тоже так зовут!"); }`
5. `var myVariable = #;`

```
if( myVariable > 5) {
alert("больше");
} else {
alert ("меньше");
}
```

Упражнение 21.2

```
<script>
var originalTitle = document.title;
function showUnreadCount( unread ) {
document.title = originalTitle + " (" + unread + " новых сообщений!");
}
showUnreadCount(5);
</script>
```