**Раздел 3.**

**WEB-программирование на стороне клиента**

**3.1 Cерверные и клиентские сценарии. Языки разработки сценариев. DHTML: JavaScript и HTML. Основы JavaScript**

**3.2 Стандартные объекты JavaScript. Создание пользовательских функций. Функции временной задержки**

**3.3 Работа с формами, слоями, изображениями**

**3.4 Библиотека JQuery: назначение и особенности использования**

**3.1 Cерверные и клиентские сценарии. Языки разработки сценариев. DHTML: JavaScript и HTML. Основы JavaScript**

Как правило, Веб-*приложение* - *приложение*, в котором клиентом выступает *браузер*, а сервером - веб-*сервер*.

Рассмотрим типы программ, обеспечивающих работу Веб и использующих HTTP-протокол.

Никакой HTTP-обмен невозможен без клиента и сервера. Однако помимо клиента и сервера в веб-сеансе могут участвовать и другие программы, которые и являются объектом *веб-программирования*.

Результатом работы веб-приложения является *веб-страница*, отображаемая в окне браузера. При этом само веб-*приложение* может выполняться как на компьютере клиента, так и на компьютере сервера.

Рассмотрим подробнее обе схемы.

### Программы, выполняющиеся на клиент-машине

Одним из типов программ, предназначенных для выполнения на клиент-машине, являются *сценарии*, например, JavaScript (VBScript). Исходный текст сценария представляет собой часть веб-страницы, поэтому *сценарий* JavaScript передается клиенту вместе с документом, в состав которого он входит. Обрабатывая *HTML*-документ, *браузер* обнаруживает исходный текст сценария и запускает его на выполнение.

Ко всем программам, которые передаются с сервера на клиент-машины и запускаются на выполнение, предъявляется одно общее требование: эти *программы должны быть лишены возможности обращаться к ресурсам компьютера, на котором они выполняются*. Такое требование вполне обосновано. Ведь передача по сети и *запуск* *Java*-апплетов и JavaScript-сценариев происходит автоматически *без участия пользователя*, поэтому работа этих программ должна быть *абсолютно безопасной для компьютера*. Другими словами, языки, предназначенные для создания программ, выполняющихся на клиент-машине, должны быть абсолютно непригодны для написания вирусов и подобных программ.

### Программы, выполняющиеся на сервере

*Код программы*, работающей на сервере, не передается клиенту. При получении от клиента специального запроса, предполагающего выполнение такой программы, *сервер* запускает ее и передает параметры, входящие в состав запроса. Средства для генерации подобного запроса обычно входят в состав *HTML*-документа.

Результаты своей работы *программа* оформляет в виде *HTML*-документа и передает их веб-серверу, а последний, в свою *очередь*, дополняет полученные данные HTTP-заголовком и передает их клиенту. Взаимодействие клиента и сервера в этом случае показано на [рисунке 5.1](http://www.intuit.ru/studies/courses/485/341/lecture/8186?page=1#image.5.1).

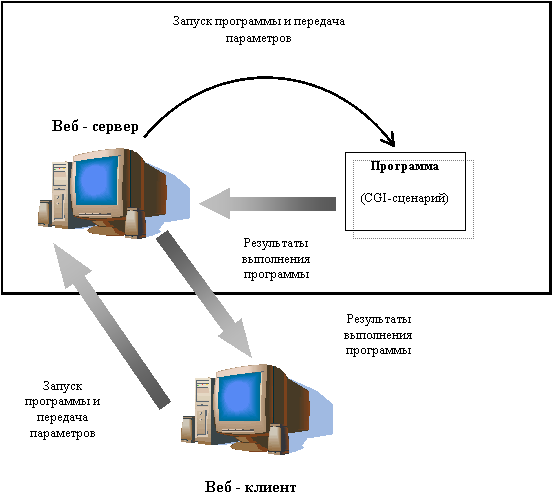


Рис.Взаимодействие клиента с программой, выполняющейся на сервере

### Насыщенные интернет-приложения

*Насыщенное интернет-приложение* (*Rich Internet application*) – еще один подход, который заключается в использовании *Adobe Flash* или *Java-апплетов* для полной или частичной реализации пользовательского интерфейса, поскольку большинство браузеров поддерживает эти технологии (как правило, с помощью *плагинов* ).

Возникновение данного подхода обусловлено тем, что в рамках веб-приложений с "тонким" клиентом взаимодействие пользователя с приложением реализуется в существенной степени через *сервер*, что требует отправки данных на *сервер*, получение ответа от сервера и перезагрузку страницы на стороне клиента.

При использовании *Java*-апплетов в состав *HTML*-документа включается специальный *дескриптор*, описывающий расположение файла, содержащего код *апплета*, на сервере. После того как клиент получает *HTML*-код документа, включающего *апплет*, он генерирует дополнительный *запрос* серверу. После того как *сервер* пересылает клиенту код *апплета*, сам *апплет* запускается на выполнение. Взаимодействие между клиентом и сервером при получении *апплета* показано на рисунке.

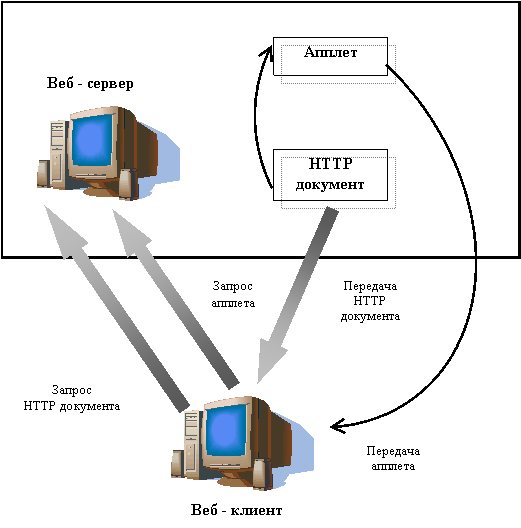


Рис. Передача клиенту Java-апплета

При использовании насыщенных интернет-приложений приходится сталкиваться со следующими проблемами:

* необходимость обеспечения безопасной среды выполнения ("песочница");
* для исполнения кода должно быть разрешено исполнение сценариев;
* потеря в производительности (т.к. выполяется на клиентской стороне);
* требуется много времени на загрузку;

### DHTML

*Динамический HTML* или *DHTML* представляет собой набор технологий, которые совместно позволяют создавать интерактивные веб-сайты на основе статического языка разметки ( *HTML* ), языка создания клиентских сценариев ( *JavaScript* ), языка описания представления документа ( *CSS* ) и документной объектной модели ( *DOM* ).

*DHTML* позволяет сценарным языкам изменять переменные языка описания представления документа, таким образом, изменяя вид и поведение прежде статического содержимого *HTML* документа уже после полной загрузки документа и в процессе просмотра его пользователем. Таким образом, динамичность, привносимая *DHTML*, проявляет себя в процессе просмотра страницы, но не имеет никакого отношения к генерации содержимого страницы при каждой ее загрузке.

В противоположность *DHTML*, *динамически генерируемая страница* - более широкое понятие, подразумевающее, например генерацию содержимого веб-страницы индивидуально для каждого пользователя. Это достигается созданием страниц с помощью клиентских или серверных (например, на *PHP* или *Perl* ) сценариев.

### Введение в JScript

JavaScript - интерпретируемый *язык программирования*, стандартизированный международной организацией *ECMA* в спецификации *ECMA-262*. Языки JavaScript, JScript и ActionScript являются расширением стандарта *ECMA-262*.

Название "*ECMAScript*" явилось фактически компромиссом между организациями, вовлеченными в процесс стандартизации, в частности Netscape и Microsoft. Хотя JavaScript и JScript стремились к совместимости с *ECMAScript*, они имеют ряд дополнительных возможностей не предусмотренных спецификацией *ECMA*.

*Синтаксис* JScript во многом аналогичен языку JavaScript, однако, помимо добавления клиентских скриптов на веб-страницы и некоторых других функций, JScript может использоваться и для других целей, например:

* автоматизация *администрирования систем* Microsoft Windows;
* создание страниц ASP.

Язык JScript получил дальнейшее развитие в виде языка JScript.*NET*, который ориентирован на работу в рамках платформы Microsoft.*NET*

JScript - интерпретируемый, объектно-ориентированный язык. Хотя он имеет существенно меньшее количество возможностей, чем такие объектно-ориентированные языки как C++ и *Java*.

Возможности языка существенно ограничены:

* язык не позволяет разрабатывать самостоятельные приложения;
* сценарии на JScript могут выполняться только при помощи интерпретатора, в частности веб-браузером.
* JScript - язык без строгого контроля типов. Поэтому не требуется объявлять тип переменных явно. Кроме того, во многих случаях JScript исполняет преобразования автоматически, когда они необходимы. Например, при *сложении строки* и числа, число будет преобразовано в строку.

Код на JScript пишется в текстовом формате, и организован в *инструкции*, *блоки*, состоящие из связанных наборов инструкций, и *комментариев*. В пределах инструкции можно использовать *переменные* и *данные*, такие как *строки*, *числа* и *выражения*. Для объявления *конца инструкции* используется точка с запятой ;. *Группа* JScript-инструкций, заключенная в *фигурные скобки* {}, называется *блоком*.

*Комментарием* в JScript является текст, расположенный после // до конца строки. *Многострочный* *комментарий* начинается с /\*, и заканчивается \*/.

Знак равенства ( = ) используется в JScript как *присваивание*. Следующий код

Pi = 3.14;

подразумевает "Присвоить *значение* 3.14 переменной Pi ".

При сравнении двух значений на *равенство* применяется двойной знак равенства ( == ).

JScript *выражения* можно разделить на *логические* или *числовые*. *Выражения* содержат некоторые особенности, к примеру, символ " + " означает "добавить к...". Любая допустимая комбинация значений, переменных, операторов, и других выражений является *выражением*.

Объявление *переменной* перед использованием является необязательным. Для этого используется *инструкция* var. *Инструкция* varявляется обязательной при объявлении локальной переменной внутри функции. Разрешается *объявление переменной* неявно - без инструкции var. Однако, в выражениях применять необъявленные переменные не допускается. JScript различает *регистр* в имени переменной. Name и name рассматриваются как различные переменные.

#### Типы данных

JScript - язык с *нестрогим контролем типов*, переменные в JScript не имеют строго фиксированного типа. *Переменные* имеют *тип*, эквивалентный типу значения, которое они содержат. Однако, в некоторых случаях, необходимо принудительное преобразование переменной в определенный тип. *Числа* могут быть объявлены как *строки*, а *строки* необходимо преобразовать в *числовой* тип. Для этого применяют функции parseInt() и parseFloat().

В JScript используется шесть типов *данных*. Основные из них - *числа*, *строки*, *объекты*, *логический*. Остальные два - null и undefined(т.е. неопределенный).

*Строки* объявляются при помощи *двойных кавычек* или *апострофов*. Строка может состоять из нуля или более символов *unicode*. Когда количество символов равно нулю, это называется пустой строкой ( "" ).

JScript поддерживает *числа* как *целые*, так и с *плавающей запятой*. Также существуют специальные представления чисел, например NaN (не число).

Примеры чисел:

3.14 // Вещественное число

26 // Целое число

0177 // Восьмеричное число

0XA8 // Шестнадцатиричное число

*Логический* тип допускает значения - true и false. Любое выражение, равное 0, считается эквивалентным false, а любое выражение, равное числу, отличному от 0 будет эквивалентным true.

Undefined – означает, что тип не определен. Значение undefined имеет переменная после ее объявления и до присвоения ей какого-либо определенного значения.

Переменная типа null - не имеет никакого определенного значения.

#### Операторы

Язык поддерживает *условные* *выражения* if и if...else. При использовании нескольких условий одновременно можно использовать операторы || (ИЛИ ) или && (И).

В JScript поддерживается несколько типов циклов: for, for...in, while, do...while и switch. Также существует инструкция *остановки* выполнения цикла. Оператор завершения break может использоваться, чтобы остановить цикл, при выполнении какого-либо условия. Инструкция continue используется, чтобы немедленно перейти к выполнению следующей итерации, пропуская остальную часть выполнения кода текущей итерации, но обновляя переменную-счетчик.

#### Функции и объекты

В JScript имеется два вида *функций*: встроенные и определяемые. Программист имеет возможность создавать собственные функции. Определение функции состоит из *объявления параметров* и *блока инструкций* JScript.

Объекты в JScript, по-сути, являются совокупностями *методов* и *свойств*. Все объекты можно разделить на три вида: *встроенные*, *созданные*и *браузерные*. Обработка *объектов* и *массивов* идентична. Можно обратиться к любой части объекта (его свойствам и методам) либо по *имени*, либо по *индексу*. Нумерация индексов в JScript начинается с нуля.

### Что такое JavaScript, и как он выполняется?

*JavaScript* является языком на основе текста, который не требует никаких преобразований перед выполнением. Другие языки, такие как *Java* и C++ требуют для выполнения компиляции, но код *JavaScript* выполняется непосредственно специальной программой, которая интерпретирует код и называется синтаксическим анализатором (почти все *браузеры* *web* содержат синтаксический анализатор *JavaScript*).

Для выполнения *JavaScript* в *браузере* имеется две возможности - либо поместить его в элемент script в любом месте внутри документа *HTML*, либо поместить его во *внешний* *файл* *JavaScript* (с расширением .js ), а затем указать этот *файл* внутри документа *HTML* с помощью пустого элемента script с атрибутом src. Оба эти метода будут рассмотрены в данном разделе.

#### Включение JavaScript в документ HTML

Самая общая форма включения JavaScript в страницу HTML будет выглядеть примерно следующим образом:

<script>

var x = 3;

alert('hello there, I am JavaScript - x is '+x);

</script>

Можно поместить этот текст в любом месте документа, и он будет выполняться, но некоторые места определенно лучше других - этот вопрос будет рассмотрен в дальнейшем.

Так как в будущем возможно использование на страницах Web нескольких различных типов сценариев, то имеет смысл добавить название используемого сценария в виде типа MIME:

<script type="text/javascript">

var x = 3;

alert('hello there, I am JavaScript - x is '+x);

</script>

**Примечание**: В Web можно встретить примеры сценариев, которые имеют атрибут language="javascript". Это не является частью какого-либо стандарта и совершенно бесполезно, удаляйте это там, где можете. Это напоминание о старых недобрых временах, когда на страницах Web был также популярен язык *VBScript*. Однако *VBScript* со временем использовать перестали, так как он работал только в Internet Explorer.

В прошлом было необходимо комментировать JavaScript в HTML, чтобы *браузер* не показывал код как HTML. Так как это относится только к очень старым *браузерам*, то об этом больше беспокоиться не стоит. Однако, если вы используете в качестве DOCTYPE строгий XHTML, необходимо помещать любой код JavaScript в блок CDATA, чтобы выполнить проверку (не беспокойтесь об этом - на самом деле это не важно на данном этапе изучения):

<script type="text/javascript">

/\* <![CDATA[ \*/

var x = 3;

alert('hello there, I am JavaScript - x is '+x);

/\* ]]> \*/

</script>

Однако для документов строгого XHTML более разумно вообще не включать какой-либо код JavaScript, а вместо этого держать его во *внешнем* документе.

#### Соединение с внешним файлом JavaScript

Чтобы соединиться с *внешним* кодом JavaScript (на том же сервере или хранящемся где-то в Сети), необходимо только добавить атрибут src в элемент script:

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

Когда такой элемент встречается на странице, *браузер* загружает файл myscript.js и выполняет его. Любое содержимое внутри самого элемента script будет пропущено, если в нем встречается атрибут src. Следующий пример загрузит файл myscript.js и выполнит код в нем, но вообще не выполнит alert внутри элемента script.

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

alert('I am pointless as I won\'t be executed');

</script>

Размещение кода JavaScript во *внешнем* файле имеет много смысла, так как:

* Можно применять один и тот же код JavaScript для нескольких документов HTML и легко осуществлять обслуживание этого кода: если потребуется внесение каких-то изменений в код, нужно будет изменить только один единственный файл.
* Код JavaScript будет кэширован *браузером*. Кэширование означает, что *браузер* будет сохранять копию кода JavaScript на компьютере посетителя сайта. Когда пользователь в следующий раз загружает тот же самый сценарий, он будет браться не с сервера, а с собственного компьютера пользователя - загружаясь, таким образом, значительно быстрее.
* Можно легко найти сценарий, если понадобится его модифицировать, избавляясь от необходимости просматривать длинные документы HTML, чтобы найти место, где требуется исправить проблему.
* Исправление ошибок становится легче, так как инструмент отладки или консоль ошибок сообщит, какой файл содержит ошибку и точно укажет номер строки.

В документ можно добавлять столько файлов JavaScript, сколько потребуется, но существует несколько соображений, которые необходимо рассмотреть, прежде чем становиться на этот путь.

### JavaScript и производительность браузера

*Разбиение* большого кода *JavaScript* на различные файлы, каждый из которых имеет дело с одной задачей, является отличной идеей для сохранения легкости обслуживания функций и ускорения исправления ошибок. Например, можно было бы иметь несколько блоков сценариев следующего вида:

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

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

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

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

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

Однако преимущества такой разработки ослабляются эффектом, который это оказывает на *производительность* документа. Разные *браузеры*ведут себя немного по разному, но в худшем варианте (который, к сожалению, реализуется в наиболее часто используемом *браузере* ) происходит следующее:

* Всякий раз, когда *браузер* встречает элемент script, он приостанавливает отображение (вывод) документа.
* Затем он загружает файл JavaScript, определенный в атрибуте src (если используется сценарий на другом сервере, то необходимо также подождать, пока *браузер* найдет этот сервер).
* Затем этот сценарий выполняется, прежде чем происходит обращение к следующему.

Все это означает, что *отображение* *web*-сайта замедляется, пока все эти действия не выполнятся для всех включенных сценариев. Это может вызывать недовольство посетителей сайта.

Одним из способов обойти эту проблему является использование вспомогательного сценария для создания единственного файла из всех используемых файлов. В этом случае сохраняется преимущество легкости обслуживания, и в то же время сокращаются задержки при отображении *web*-страницы. В *Web* существует несколько подобных сценариев - один из них написан на *PHP* и доступен у *Ed* Eliot .

Задержка отображения также определяет, куда желательно поместить *JavaScript* в документе.

### Куда поместить JavaScript

Технически *JavaScript* можно поместить в любом месте документа. Решение, которое необходимо сделать, состоит в сопоставлении *производительности* и облегчения для разработчиков поиска сценариев, и обеспечение незамедлительной работы для посетителей *JavaScript*.

При классическом подходе сценарии помещают в заголовке head документа:

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"

"http://www.w3.org/TR/html4/strict.dtd">

<html lang="en-en">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<title></title>

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

</head>

<body>

<!-код HTML здесь ->

</body>

</html>

Такой подход имеет преимущество, обеспечивая предсказуемое местоположение сценариев - разработчики знают, где их искать. Он также имеет преимущество гарантии того, что весь код *JavaScript* загружается и выполняется до отображения документа.

Недостатки состоят в том, что сценарии задерживают *отображение* документа, и что *сценарий* не имеет доступа к *HTML* в документе. Поэтому требуется задержка выполнения любых сценариев, которые изменяют *HTML* документа, пока документ не завершит загрузку. Это можно сделать с помощью обработчика событий **onload** или одного из различных или **contentAvailable** существующих в *Web* - ни одно из которых не является полностью надежным и большинство использует специфические особенности *браузеров*.

Специалисты по *производительности* в последнее время стали предлагать вместо этого *размещение* *JavaScript* в конце body:

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"

"http://www.w3.org/TR/html4/strict.dtd">

<html lang="en-en">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<title></title>

</head>

<body>

<!-код HTML здесь ->

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

</body>

</html>

Преимущество в том, что *JavaScript* не задерживает *отображение* *HTML*, а также, что уже доступен весь код *HTML*, который желательно изменить с помощью *JavaScript*.

Один из недостатков в том, что такая практика еще не слишком распространена, и можно запутать разработчиков, которые поддерживают код. Другой - более проблематичный - недостаток в том, что *HTML* становится доступен до загрузки *JavaScript*. Хотя именно это мы и хотим получить, но это также означает, что посетители сайта начинают взаимодействовать с интерфейсом до того, как появляется возможность его изменить. Пусть, например, желательно проверить форму с помощью *JavaScript*, прежде чем она отправляется на *сервер* - форма может быть отправлена до того как *сценарий* будет загружен. Если *сценарий* написан с целью простого усовершенствования (и не создает каких-либо зависимостей), то это не проблема - просто источник недовольства.

Поэтому необходимо выбирать, что в большей степени подходит целям *web*-сайта, можно даже использовать смесь обоих подходов - поместить наиболее важные сценарии в head, и вызывать их в сочетании со сценариями "хорошо бы иметь" в конце документа.

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"

"http://www.w3.org/TR/html4/strict.dtd">

<html lang="en-en">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<title></title>

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

</head>

<body>

<!-код HTML здесь ->

<script type="text/javascript">

applyFunctionality();

</script>

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

</body>

</html>

В любом случае необходимо проверить, что сценарии представлены в правильном порядке, так как *браузеры* будут загружать и анализировать их один за другим. Это также ведет к еще одной проблеме, которую необходимо рассматривать при использовании*JavaScript*.

### Безопасность JavaScript и ее отсутствие

Невозможно переоценить этот вопрос. *JavaScript* является прекрасным языком и может помочь в создании быстродействующих и привлекательных интерактивных *Web*-сайтов и приложений, но крайне слабым местом его является *безопасность*. Вкратце можно сказать, что в *JavaScript* отсутствует модель *безопасности*, и с его помощью вы не можете защищать, шифровать, охранять или хранить что-то жизненно важное или секретное.

Каждый *сценарий* на странице обладает одинаковыми правами - все они могут обращаться друг к другу, считывать переменные, получать *доступ* к функциям, и также переопределять друг друга. Если в первом включенном сценарии имеется *функция* с именем init(), и другая *функция* в последнем включенном сценарии, то исходная *функция* будет переопределена. Мы вернемся к рассмотрению этой проблемы в следующей лекции этого курса.

Все это не было бы большой проблемой, если бы никогда не использовались сценарии других людей. Однако, поскольку большая часть *Интернет* рекламы и статистики слежения сделана с помощью *JavaScript*, это вряд ли возможно - постоянно приходится использовать сценарии сторонних разработчиков.

Сценарии могут также считывать cookies и используя prototype функции, можно переопределить любую встроенную функцию *JavaScript*. Наконец, *JavaScript* можно легко отключить, поэтому можно забыть о том, что защита с помощью *JavaScript* является хорошим средством *безопасности*.

Код *JavaScript* всегда легко доступен другим разработчикам для чтения и анализа. Конечно, можно упаковать (удалить все ненужные пробелы) и запутать (использовать случайные имена переменных и функций) свои сценарии, но даже их можно легко декодировать, поэтому заставить прекратить использовать ваш собственный код можно только самого себя. Доступность исходного кода и возможность чтения и анализа его является, возможно, основным фактором успеха *JavaScript* - в течение многих лет мы учились, заглядывая в решения других людей. К счастью теперь это прошло, так как стали доступны хорошие книги и учебники.

Хотя упаковка и запутывание будут бесполезны в качестве средств *безопасности*, они часто используются для сценариев среднего и большого размера, перед тем как разместить их в *Web*, как часть процесса публикации. Это помогает уменьшить полосу пропускания, необходимую для представления сайта пользователям. Экономия нескольких байтов там и здесь может показаться несущественной для какого-нибудь блога о котятах, но это может оказаться существенным, когда речь идет о сайтах масштаба *google*.*com*.

### Методы, которые желательно избегать

Наибольшая проблема при изучении *JavaScript* состоит в том, что существует огромный объем устаревшей и возможно опасной информации. Это особенно печально, так как большая часть этой информации очень хорошо представлена и создает у большого числа новичков ложное чувство знания *JavaScript*, когда они копируют какой-либо готовый код.

Так как среда, в которой применяется *JavaScript*, по большей части неизвестна (пользователи могут иметь любую настройку), и мы не знаем, какие решения привели к найденному в *Web* коду, созданному определенным образом, мы не будем указывать на конкретные решения. Однако следующие идеи являются идеями из прошлого и их необходимо использовать только как последнее средство для поддержки действительно старых *браузеров*.

document.write() - можно вывести содержимое документа с помощью document.write(), но при этом возникает несколько проблем: необходимо смешивать код *HTML* и *сценарий* и добавлять узел сценария точно в том месте, где должно появится содержимое, что замедляет работу страницы. Это очень привлекательный способ быстро показать (например, в учебнике или при тестировании/отладке кода), каким будет результат фрагмента кода, но это плохой пример, чтобы показывать его людям, так как это никогда не должно использоваться в реальном коде.

<noscript></noscript> - как предполагает название, элемент noscript является противоположностью элемента script. Содержимое внутри него будет показано пользователям, которые отключили *JavaScript*. Основная причина использования noscript состоит в предоставлении альтернативного контента для пользователей, которые не имеют доступного *JavaScript* в *браузере*. Пользователи, которые не имеют *JavaScript*, делают это не для того, чтобы усложнить жизнь разработчикам - они могут делать это по соображениям политики *безопасности*, или потому что используемый *браузер* не поддерживает *JavaScript*. Но внутри noscript можно представить не так много информации, и, конечно, не стоит использовать это для указания пользователю включить в *браузере* *JavaScript* - для некоторых пользователей это невозможно.

<a href="javascript:doStuff()">…</a> - это был очень распространенный способ вызвать функцию *JavaScript*, чаще всего, когда кнопки были недоступны (в старых *браузерах* невозможно создавать кнопки). Проблема в том, что это некорректная *ссылка*, так как *javascript* не является протоколом *интернет* (как *ftp*:// или http://). Если *интерпретатор* *JavaScript* отключен, *ссылка* все равно появляется и создает у пользователя ложную надежду, что что-то произойдет.

document.layers и document.all - оба эти решения были эквивалентами *DOM* в старых *браузерах* (Netscape 4.x и *Internet* Explorer 4, соответственно) и если только вам не требуется их поддерживать (простите, если вы вынуждены), этот код совершенно лишний.

**3.2 Стандартные объекты JavaScript. Создание пользовательских функций. Функции временной задержки**

Язык JavaScript является объектно-ориентированным языком. Однако его объектная модель отличается от общеизвестных моделей таких языков программирования как Java и C++ тем, что в нем используется модель прототипов, а не классов. В языке JavaScript нет механизма классов, все объекты создаются на основе прототипа. Поэтому объектная модель JavaScript проще, но в связи с этим она и не столь мощна, как модели, основанные на классах. В данном языке нет ни наследования, ни полиморфизма.

Любой объект в **JavaScript** создается конструктором этого объекта - специальной функцией, в которой определяются свойства и методы объекта. Для встроенных объектов конструкторы реализованы в виде функций, написанных на низкоуровневом языке процессора и встроенных в язык **JavaScript**. Для создания объекта необходимо выполнить операцию **new** с единственным операндом - конструктором этого объекта. Результатом выполнения этой операции будет выделение области оперативной памяти для хранения объекта, его инициализация и возврат ссылки на выделенную область памяти, которую можно сохранить в переменной для последующего обращения к объекту. Все сказанное относится и к создаваемым пользователем объектам, но конструктор пишется на языке JavaScript с применением специальных правил создания свойств и методов объекта.

### Объект Array

Массив - упорядоченный набор однородных данных, к элементам которого можно обращаться по имени и индексу. В **JavaScript**, так как в нем отсутствует определение переменных определенного типа, это определение расширяется до хранения не обязательно однородных данных.

Массив, как и любой объект, создается с помощью операции new и своего конструктора - функции **Array**. Как и в других объектно-ориентированных языках, у объекта определенного типа может быть несколько конструкторов, отличающихся друг от друга только передаваемыми в него параметрами (имя всегда остается одно и то же):

a = new Array() // пустой массив (количество элементов 0)

a = new Array(число) // массив с заданным числом элементов

a = new Array(элемент0, элемент1, . . . , элементN);

Количество элементов может быть равно максимальному значению 4-х байтного числа без знака.

Для получения значения элемента массива следует в квадратных скобках рядом с именем массива указать порядковый номер элемента (выполнить операцию индексацию []):

a[0] = "1"; a[1] = 2;

c = b[2]\*c[3];

Нумерация элементов массива начинается с 0. При попытке получить значение элемента массива с несуществующим индексом никакой ошибки генерироваться не будет, а всегда будет возвращено значение *undefined*.

|  |  |
| --- | --- |
| **Методы** | **Действие** |
| concat([item1[, item2[, . . . [,itemN]]]]) | Возвращает массив, представляющий собой массив, метод которого вызывается, с добавленными в его конец элементами, заданными в качестве параметров. Если параметр сам является массивом, то добавляются все его элементы. Сам массив не изменяется. |
| join(separator) | Возвращает строку - результат конкатенации всех элементов массива. Соседние элементы в строке разделены указанным в качестве параметра разделителем separator. |
| **Методы** | **Действие** |
| pop() | Удаляет последний элемент из массива и возвращает его значение. |
| push([item1[, item2[, . . . [,itemN]]]]) | Добавляет элементы в конец массива и возвращает последний добавленный элемент. Если добавляемый элемент массив, то добавляет его как один элемент. |
| reverse() | Переставляет элементы массива в обратном порядке: первый элемент становится последним, а последний первым. |
| shift() | Удаляет первый элемент массива и возвращает его значение. |
| slice(start [, end]) | Возвращает массив, содержащий часть исходного массива (сечение). Если параметр end не определен, то возвращаются все элементы от элемента с индексом start до конца массива. |
| splice(start, deleteCount, [item1[, item2[, … [,itemN]]]]) | Удаляет заданное вторым параметром deleteCount количество элементов из массива, начиная с элемента с индексом start. Если заданы необязательные параметры item, то добавляет их вместо удаленных. Возвращает массив удаленных элементов. |
| sort([имя\_функ]) | Возвращает массив, элементы которого отсортированы заданным в функции методом. Если метод сортировки вызывается без параметра, то элементы сортируются по порядку возрастания номеров начальных символов в таблице ASCII. Если при вызове метода параметром указывается ссылка на функцию сортировки задана, то она должна быть написана в соответствии со следующей спецификацией: она имеет два обязательных параметра и возвращает отрицательное значение, если значение первого параметра меньше значения второго, ноль, если они равны и положительное значение, если значение первого параметра превосходит значение второго. |
| unshift([item1[, item2[,… [,itemN]]]]) | Добавляет один или более элементов в начало массива и возвращает новую длину массива. |
| toString() | Возвращает строку - результат конкатенации всех элементов массива. Элементы массива в строке разделены запятой. |

Несколько примеров использования методов объекта массив.

Пусть определены два массива:

array1 = new Array("Первый","Второй","Третий");

array2 = new Array("Один","Два","Три");

Тогда метод join() первого массива array1.join() возвратит строку:

*"Первый,Второй,Третий"*

А его же метод array1.join(":") возвратит следующую строку:

*"Первый:Второй:Третий"*

Заметим, что метод toString() массива (наследуется от встроенного объекта Object и переопределяется в конструкторе **String**()) , вызываемый по умолчанию тогда, когда надо представить объект-массив в виде строки, по-существу вызывает его метод join() без параметра. Например, если необходимо отобразить в документе содержимое массива, то можно использовать следующий оператор:

document.write("<p>", array1, "</p>");

Результатом будет новый абзац со значениями массива array1, разделенными запятыми:

*"Первый,Второй,Третий"*

Метод sort() первого массива array1.sort() упорядочит элементы массива array1 (переставив их местами непосредственно в самом массиве array1) в алфавитном порядке:

*array1[0] = "Второй"*

*array1[1] = "Первый"*

*array1[2] = "Третий"*

Так как некоторые методы массива возвращают массив, то к нему можно сразу же применить какой-либо метод, продолжив "точечную" нотацию. Например,

array1.concat(array2).sort()

объединит два массива в один новый и отсортирует его.

Массив, как объект, имеет ***свойство******length***, хранящее количество элементов массива. Добавление или удаление элементов массива (методы pop, push, shift, unshift, splice) приводит к автоматическому изменению свойства length.

Добавить в конец массива элемент можно не только с помощью методов push или splice, но и простым заданием элемента с индексом, большим индекса его последнего элемента. При этом все промежуточные элементы принимают значения *undefined*.

Массивы JavaScript "понимают" не только целые числовые индексы, но и индексы, заданные в виде строки. Однако следует иметь в виду, что добавление в массив элементов со строковыми индексами не приводит к увеличению значения свойства length, а это означает, что подобные "элементы" не участвуют ни в каких методах массива. При создании элемента со строковым индексом на самом деле создается новое свойство конкретного объекта-массива.

Для массивов нескольких размерностей конструктор не предусмотрен. Но их можно создавать, объявляя каждый элемент одномерного массива снова одномерным массивом. Полученные элементы уже двумерного массива можно снова объявить как одномерные массивы, создав, таким образом, трехмерный массив и т. д. Следующий код создает двумерный массив размерности (4 4):

a = new Array()

for (i=0; i < 4; i++) {

a[i] = new Array()

for (j=0; j < 4; j++) {

a[i][j] = значение

}

}

Обратите внимание, что при ссылке на элементы такого массива индекс каждого размерения элемента следует задавать в самостоятельных квадратных скобках: сначала в квадратных скобках указать первый индекс, затем в других квадратных скобках второй индекс и т. д.

### Объект Math

В отличие от других объектов этот объект создавать не надо - он создается интерпретатором автоматически при загрузке документа со сценарием в браузер. Поэтому обращение к его методам и свойства выполняются с использованием его имени **Math**:

Math.PI // число π

Math.abs(x) // вычисление абсолютного значения

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| abs(x) | Абсолютное значение |
| sin(x), cos(x), tan(x) | Стандартные тригонометрические функции; аргумент задается в радианах |
| acos(x), asin(x), atan(x) | Обратные тригонометрические функции |
| exp(x), log(x) | Экспоненциальная функция и функция натурального логарифма |
| ceil(x) | Наименьшее целое, большее или равное значению аргумента |
| floor(x) | Наибольшее целое, меньшее или равное значению аргумента |
| min(x,y), max(x,y) | Наибольшее или наименьшее значение двух аргументов |
| pow(x,y) | Показательная функция: pow(x,y) = xy |
| round(x) | Округление аргумента до ближайшего целого |
| sqrt(x) | Квадратный корень |
| atan2(y,x) | Возвращает угол в радианах между осью Х и направлением на точку (х, у) |
| random() | Возвращает случайное число между 0 и 1. Генератор случайных чисел запускается автоматически при загрузке ядра JavaScript |

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| E | Возвращает постоянную Эйлера e - основание натурального логарифма. |
| LN2 | Возвращает натуральный логарифм числа 2. |
| LN10 | Возвращает натуральный логарифм числа 10. |
| LOG2E | Возвращает логарифм по основанию 2 постоянной Эйлера e. |
| LOG10E | Возвращает логарифм по основанию 10 постоянной Эйлера e. |
| PI | Число π. |
| SQRT1\_2 | Возвращает квадратный корень из 0.5. |
| SQRT2 | Возвращает квадратный корень из 2. |

### Объект String

Когда переменной присваивается строковый литерал, она становится строковой переменной. На самом деле JavaScript создает встроенный объект **String**. Таким образом, любая строковая переменная или строковый литерал является объектом **String**, к которому могут быть применены все методы этого объекта.

Хотя обычно в сценарии строка создается присваиванием переменной строкового литерала (а в этом случае создается и объект **String**), в JavaScript предусмотрен конструктор для этого типа объекта:

имя\_объекта = new String(строка);

Параметром конструктора является строковый литерал или переменная:

myString = new String("Строка");

a = 2.67;

myNumber = new String(a); // myNumber = "2.67"

Объект **String** имеет свойства

*length* - задает длину строки, содержащейся в строковом объекте, т. е. количество символов в строке;

*prototype* - позволяет создавать новые свойства и методы, связанные с объектом **String**.

И "Строка".length, и myString.length возвращают одинаковые значения 6, равные в первом случае длине строкового литерала, а во втором случае длине строки, содержащейся в строковом объекте.

Объект **String** имеет два типа методов: первые возвращают отформатированный HTML-вариант строки, а вторые выполняют некоторые действия над содержимым строки.

Методы, возвращающие HTML-отформатированные варианты строк, возвращают строки, заключенные в открывающий и закрывающий теги определяемого методом элемента HTML. Например, следующий оператор вставляет в страницу HTML гиперсвязь с ресурсом, расположенным по адресу, задаваемому параметром метода link строки:

document.write(s.link("http://www.altavista.com"));

В документе отобразится содержимое строкового объекта **s**, представленное как гиперссылка на ресурс, заданный параметром метода.

Методы объекта **String**

|  |  |
| --- | --- |
| anchor(name) | Возвращает строку, заключенную в теги HTML <A> и </A>, и устанавливает атрибут name элемента A равным значению параметра name. |
| big() | Возвращает строку, заключенную в теги HTML <BIG> и </BIG>. |
| blink() | Возвращает строку, заключенную в теги HTML <BLINK> и </BLINK>. |
| bold() | Возвращает строку, заключенную в теги HTML <B> и </B>. |
| fixed() | Возвращает строку, заключенную в теги HTML <TT> и </TT>. |
| fontcolor(colorValue) | Возвращает строку, заключенную в теги HTML <FONT> и </FONT>, и устанавливает атрибут COLOR этого тега равным значению параметра colorValue. |
| fontSize(sizeValue) | Возвращает строку, заключенную в теги HTML <FONT> и </FONT>, и устанавливает атрибут SIZE равным значению параметра sizeValue. |
| italics() | Возвращает строку, заключенную в теги HTML <I> и </I>. |
| link(href) | Возвращает строку, заключенную в теги HTML <A> и </A>, и устанавливает атрибут HREF элемента A равным значению параметра href. |
| small() | Возвращает строку, заключенную в теги HTML <SMALL> и </SMALL>. |
| strike() | Возвращает строку, заключенную в теги HTML <STRIKE> и </STRIKE>. |
| sub() | Возвращает строку, заключенную в теги HTML <sub> и </sub>. |
| sup() | Возвращает строку, заключенную в теги HTML <sup> и </sup>. |
| charAt(index) | Возвращает символ, находящийся на указанном месте (с индексом равным index) в строке. Индексы начинаются с 0. |
| charCodeAt(index) | Возвращает код символа (Unicode), расположенного на указанном месте (с индексом равным index) в строке. Если в строке нет символа с указанным индексом, возвращается значение NaN. Индексы в строке начинаются с 0. |
| concat(string2) | Возвращает строку, представляющую собой результат объединения двух строк. |
| String.fromCharCode( char1, char2, ... ) | Возвращает строку, составленную из указанных символов, заданных кодом Unicode. Количество параметров не ограничено. |
| indexOf(substring, startindex) | Возвращает целое число - позицию первого вхождения подстроки substring в строку, начиная с позиции startindex. Если такой подстроки не найдено, возвращает -1. |
| lastIndexOf(substring, startindex) | Возвращает целое число - позицию последнего вхождения подстроки substring в строку, начиная с позиции startindex. Если такой подстроки не найдено, возвращает -1. |
| localCompare(string) | Сравнение строк в соответствии с локальными языковыми установками: -1 строка раньше параметра, 1 наоборот и 0 строки равны. |
| match(regExpression) | Возвращает массив, состоящий из символов, соответствующих регулярному выражению regExpression. |
| replace(regExpr,replaceString) | Возвращает строку, в которой фрагменты исходной строки, соответствующие образцу регулярного выражения regExpression, заменены на строку replaceString. |
| search(regExpression) | Возвращает позицию первой подстроки, соответствующей регулярному выражению. Если такая строка не найдена, то возвращает -1. |
| slice(start[, end]) | Возвращает часть строки, начиная с позиции start и заканчивая позицией end. Если аргумент end отсутствует, то возвращается строка от start до самого конца. |
| split(separator) | Возвращает массив строк, созданный из подстрок, разделенных разделителем separator. Разделитель также может являться регулярным выражением. |
| substr(start[,length]) | Возвращает подстроку данной строки, которая начинается с символа в позиции start и имеет указанную длину length. Если длина не указана, то возвращается вся строка, начиная с позиции start. Если позиция символа превосходит число символов в строке, то возвращается пустая строка. |
| substring(start, end) | Возвращает подстроку данной строки, заданной начальной и конечной позициями. Наименьшее из двух заданных значений используется в качестве начальной позиции выделяемой подстроки. Если начальная позиция превосходит число символов в строке, то возвращается пустая строка. |
| toLowerCase() | Возвращает строку, в которой все буквенные символы преобразованы в строчные. |
| toUpperCase() | Возвращает строку, в которой все буквенные символы преобразованы в прописные. |
| toString() | Преобразует объект в строку. |

### Объект Date

Объект **Date** позволяет манипулировать с датами и временем в программах, написанных на **JavaScript**. Время в **JavaScript** измеряется в миллисекундах, т.е. в тысячных долях секунды: Отсчет производится **с 1 января 1970 года**. Даты до этого времени будут измеряться в миллисекундах со знаком минус.

##### Создание объекта Date

Перед работой с датой или временем нужно создать копию объекта **Date**. Это достигается вызовом функции (конструктора) **Date**() с ключевым словом **new.** Например,

my\_date=new Date();

При этом дата и время считываются с пользовательского компьютера и сохраняются в переменной **my\_date.**

Создать и определить новый объект можно, используя аргументы функции **Date**. Например, 31 января 2000 года, 00:20:00 по Гринвичу.

my\_date=new Date(949278000000); **--** в этом случае задано число миллисекунд, прошедших с 0 часов 1 января 1970 года. В таком виде **JavaScript** хранит даты.

my\_date=new Date(2000,0,1); **--** задан год, месяц и день.

my\_date=new Date(2000,0,1,0,0,0); **--** задан год, месяц и день, а также часы, минуты и секунды.

my\_date=new Date(“31 January 2000”); **--** дата задана в виде текстовой строки.

my\_date=new Date(“January 31, 2000”); **--** дата задана в виде текстовой строки.

##### Методы объекта Date

|  |  |
| --- | --- |
| getDate() | Возвращает день месяца даты, хранящейся в объекте date. Значения от 1 до 31. |
| getDay() | Возвращает число, соответствующее дню недели даты, хранящейся в объекте date. 0 соответствует воскресенью, 6 - субботе. |
| getFullYear() | Возвращает год в четырехзначном варианте даты, хранящейся в объекте date. |
| getHours() | Возвращает час в 24-часовом формате даты, хранящейся в объекте date. Если при задании даты час не был указан, то этот метод возвращает 0. |
| getMilliseconds() | Возвращает миллисекунды даты, хранящейся в объекте date. Если при задании даты миллисекунды не были указаны, то этот метод возвращает 0. |
| getMinutes() | Возвращает минуты даты, хранящейся в объекте date. Если при задании даты минуты не были указаны, то этот метод возвращает 0. |
| getMonth() | Возвращает номер месяца даты, хранящейся в объекте date. Январю соответствует номер 0, декабрю - номер 11. |
| getSeconds() | Возвращает секунды даты, хранящейся в объекте date. Если при задании даты секунды не были указаны, то этот метод возвращает 0. |
| getTime() | Возвращает целое число, равное количеству миллисекунд, прошедшее между датой, хранящейся в объекте date, и 1 января 1970 года по Гринвичу. Отрицательные значения соответствуют датам до 1 января 1970 года. |
| getTimeZoneOffset() | Возвращает разницу во времени между местным временем и по Ггринвичу в минутах. Значение является целым числом. |
| getUTCDate() | Возвращает день месяца даты, хранящейся в объекте date. Значения от 1 до 31. Используется формат UTC. |
| getUTCDay() | Возвращает число, соответствующее дню недели даты, хранящейся в объекте date. 0 соответствует воскресенью, 6 - субботе. Используется формат UTC. |
| getUTCFullYear() | Возвращает год в четырехзначном варианте даты, хранящейся в объекте date. Используется формат UTC. |
| getUTCHours() | Возвращает час в 24-часовом формате даты, хранящейся в объекте date. Если при заданиидаты час не был указан, то этот метод возвращает 0. Используется формат UTC. |
| getUTCMilliseconds() | Возвращает миллисекунды даты, хранящейся в объекте Date. Если при задании даты миллисекунды не были указаны, то этот метод возвращает 0. Используется формат UTC. |
| getUTCMinutes() | Возвращает минуты даты, хранящейся в объекте date. Если при задании даты минуты не были указаны, то этот метод возвращает 0. Используется формат UTC. |
| getUTCMonth() | Возвращает номер месяца даты, хранящейся в объекте date. Январю соответствует номер 0, декабрю - номер 11. Используется формат UTC. |
| getUTCSeconds() | Возвращает секунды даты, хранящейся в объекте date. Если при задании даты секунды не были указаны, то этот метод возвращает 0. Используется формат UTC. |
| getVarDate() | Возвращает значение объекта Date в формате VT\_DATE. Используется только в объектах ActiveX, которые поддерживают такой формат. |
| setDate(dateValue) | Устанавливает день месяца для объекта Date. Если переданное в метод значение больше количества дней в месяце даты, хранящейся в объекте Date, то хранящаяся в объекте дата заменяется датой, день месяца которой получается вычитанием из переданного значения параметра количества дней хранимого месяца, а месяц переходит на следующий после хранимого (этот алгоритм повторяется до тех пор, пока не будет получено правильное число месяца). Если переданное значение отрицательное, то дата меняется на дату, отстоящую от начала хранимого месяца на число дней, соответствующее абсолютному значению переданного параметра. Пример: если текущее значение даты в объекте Date равно 3 Мая 2000, и вызван метод setDate(33), то дата поменяется на 2 Июня 2000. Если вызван метод setDate(-1), то то дата поменяется на 29 Апреля 2000. |
| setFullYear  (yearValue[, monthValue,  dayValue]) | Устанавливает год для объекта Date. При желании вы можете использовать этот метод для задания значений дня и месяца. |
| setHours(hoursValue [, minutesValue, secondsValue, millisecondValue]) | Устанавливает час для объекта Date. При желании вы можете использовать этот метод для задания значений минут, секунд и миллисекунд. Если указанное значение часа превосходит количество часов в сутках (больше 24) или отрицательно, то соответственно меняется дата. Пример: если текущие дата и время в объекте Date равны 3 Мая 2000 00:00:00:00, и вызван метод setHours(25), то результатом будет 4 Мая 2000 01:00:00:00. Отрицательный аргумент уменьшает текущее значение даты на количество часов, равное абсолютному значению переданного параметра, относительно начала суток даты, хранящейся в объекте Date. |
| setMilliseconds( [millisec] ) | Устанавливает значение миллисекунд для объекта Date. Если аргумент превосходит 999 или отрицателен, то значения секунд, минут, часов и т.д. по мере необходимости увеличиваются или уменьшаются на соответствующее число. |
| setMinutes( minutesValue [,secondsValue,millisecondValue]) | Устанавливает значение минут для объекта Date. Этот метод можно использовать также для установления значений секунд и миллисекунд. Если аргумент превосходит 59 или отрицателен, то значения часов, дня, месяца и т.д. по мере необходимости увеличиваются или уменьшаются на соответствующее число. |
| setMonth( monthValue [,dayValue]) | Устанавливает месяц для объекта Date. Январю соответствует номер 0, декабрю - номер 11. Метод также может быть использован для установления значения дня месяца. Если аргумент превосходит 11 или отрицателен, то год хранимой даты увеличивается или уменьшается на соответствующее число. |
| setSeconds( secondsValue [,millisecondValue]) | Устанавливает значение секунд для объекта Date. Этот метод можно использовать также для установления значений миллисекунд. Если аргумент превосходит 59 или отрицателен, то значения минут, часов, дня, месяца и т.д. по мере необходимости увеличиваются или уменьшаются на соответствующее число. |
| setTime( milliseconds ) | Устанавливает дату и время для объекта Date. Указанный аргумент должен являться целым числом, равным количеству миллисекунд между указанной датой и 1 января 1970 года UTC. Отрицательные значения соответствуют датам до 1 января 1970 года. |
| setUTCDate( dateValue ) | Устанавливает день месяца для объекта Date. Если переданное в метод значение больше количества дней в месяце даты, хранящейся в объекте Date, то хранящаяся в объекте дата заменяется датой, день месяца которой получается вычитанием из переданного значения параметра количества дней хранимого месяца, а месяц переходит на следующий после хранимого (этот алгоритм повторяется до тех пор, пока не будет получено правильное число месяца). Если переданное значение отрицательное, то дата меняется на дату, отстоящую от начала хранимого месяца на число дней, соответствующее абсолютному значению переданного параметра. Пример: если текущее значение даты в объекте Date равно 3 Мая 2000, и вызван метод setDate(33), то дата поменяется на 2 Июня 2000. Если вызван метод setDate(-1), то то дата поменяется на 29 Апреля 2000. Используется формат UTC. |
| setUTCFullYear( yearValue[, monthValue, dayValue] ) | Устанавливает год для объекта Date. При желании вы можете использовать этот метод для задания значений дня и месяца. Используется формат UTC. |
| setUTCHours(hoursValue [,minutesValue, secondsValue, millisecondValue]) | Устанавливает час для объекта Date. При желании вы можете использовать этот метод для задания значений минут, секунд и миллисекунд. Если указанное значение часа превосходит количество часов в сутках (больше 24) или отрицательно, то соответственно меняется дата. Пример: если текущие дата и время в объекте Date равны 3 Мая 2000 00:00:00:00, и вызван метод setHours(25), то результатом будет 4 Мая 2000 01:00:00:00. Отрицательный аргумент уменьшает текущее значение даты на количество часов, равное абсолютному значению переданного параметра, относительно начала суток даты, хранящейся в объекте Date. Используется формат UTC. |
| setUTCMilliseconds( millisec ) | Устанавливает значение миллисекунд для объекта Date. Если аргумент превосходит 999 или отрицателен, то значения секунд, минут, часов и т.д. по мере необходимости увеличиваются или уменьшаются на соответствующее число. Используется формат UTC. |
| setUTCMinutes(minutesValue [,secondsValue, millisecondValue]) | Устанавливает значение минут для объекта Date. Этот метод можно использовать также для установления значений секунд и миллисекунд. Если аргумент превосходит 59 или отрицателен, то значения часов, дня, месяца и т.д. по мере необходимости увеличиваются или уменьшаются на соответствующее число. Используется формат UTC. |
| setUTCMonth(monthValue [,dayValue]) | Устанавливает месяц для объекта Date. Январю соответствует номер 0, декабрю - номер 11. Метод также может быть использован для установления значения дня месяца. Если аргумент превосходит 11 или отрицателен, то год увеличивается или уменьшается на соответствующее число. Используется формат UTC. |
| setUTCSeconds( secondsValue[, millisecondValue]) | Устанавливает значение секунд для объекта Date. Этот метод можно использовать также для установления значений миллисекунд. Если аргумент превосходит 59 или отрицателен, то значения минут, часов, дня, месяца и т.д. по мере необходимости увеличиваются или уменьшаются на соответствующее число. Используется формат UTC. |
| toGMTString() | Возвращает дату, преобразованную в строку в формате GMT (Greenwich Mean Time - время по Гринвичу). Пример: 03 Мая 2000 00:00:00:00 GMT. |
| toLocaleString() | Возвращает дату, преобразованную в строку. Время и дата отображается в формате, заданном на конкретном компьютере, где выполняется код. |
| toUTCString() | Возвращает дату, преобразованную в строку. Используется формат UTC. |
| valueOf() | Возвращает целое число, равное количеству миллисекунд между указанной датой и 1 января 1970 года UTC. Отрицательные значения соответствуют датам, предшествующим 1 января 1970 г. |

В приводимой ниже таблицы представлены форматы отображения даты некоторыми методами объекта **Date**:

|  |  |
| --- | --- |
| **Метод** | **Результат** |
| toString() | Sun Sep 14 21:35:08 UTC+0400 2003 |
| toLocaleString() | воскресенье, 14 сентября 2003 г. 21:35:08 |
| toGMTString() | Sun, 14 Sep 2003 17:35:08 UTC |
| toUTCString() | Sun, 14 Sep 2003 17:35:08 UTC |
| toDateString() | Sun Sep 14 2003 |
| toLocaleDateString() | воскресенье, 14 сентября 2003 г. |
| toTimeString() | 21:35:08 UTC+0400 |
| toLocaleTimeString() | 21:35:08 |

### Объект Function

Любая функция в **JavaScript** является объектом типа **Function**. Создать функцию можно не только конструкцией function, но и с помощью конструктора **Function**:

var имя\_функции = new Function([argname1, [... argnameN,]] тело\_функции)

Все параметры являются строками. Сначала определяется список формальных параметров, а потом строка, содержащая операторы тела функции.

В **JavaScript** часто используется тот факт, что функция является объектом. Если вспомнить вызов метода сортировки массива, то, по-существу, параметром в нем является ссылка на функцию. Так что избегайте записывать вызов функции (или метода объекта) без круглых скобок. В **JavaScript** запись "вызова" функции без круглых скобок не является синтаксической ошибкой, но всего лишь ссылка на объект **Function**, которая часто используется для задания функции обработки событий объектов, соответствующих элементам HTML отображаемого в браузере документа.

##### Свойства объекта Function

|  |  |
| --- | --- |
| аrguments | Объект, хранящий переданные в функцию фактические параметры. Он не является массивом, но доступ к хранящимся в нем фактическим параметрам осуществляется как к элементу массива с использованием целочисленного индекса, который изменяется от нуля и далее. |
| caller | Возвращает ссылку на объект Function, представляющий функцию, вызвавшую исходную. Если свойство имеет значение null, то функция вызвана из верхнего уровня сценария, а не из другой функции. |
| callee | Возвращает ссылку на саму себя. Является свойством объекта arguments функции, а не самой функции. |
| length | Возвращает количество обязательных параметров, которые следует передавать в функцию при ее вызове, т. е. тех параметров, которые определены в объявлении функции. |

Все свойства объекта **Function** становятся доступными только тогда, когда функция, определяемая этим объектом, выполняется. Поэтому доступ к ним возможен только из тела функции.

### Объект Boolean

Является объектным интерфейсом для типа данных **Boolean**. Создается неявным образом, когда булевый тип преобразуется к объекту, но может быть создан явно с помощью конструктора Boolean():

var b = new Boolean([параметр]);

В случае отсутствия параметра, или если он равен **0, NaN, false, null** или пустой строке **""**, значение объекта есть ложь, иначе истина.

Объект **Boolean** используется достаточно редко.

### Объект Number

Является объектным интерфейсом для числового типа данных. Создается неявным образом, когда числовой тип преобразуется к объекту, но может быть создан явно с помощью конструктора **Number**():

var b = new Number(числовое\_значение);

|  |  |
| --- | --- |
| NaN | Специальное числовое значение, указывающее на то, что вычисление арифметического выражения не привело к числовому результату. |
| MAX\_VALUE, MIN\_VALUE | Максимальное число и минимальное число, представимые в JavaScript. |
| POSITIVE\_INFINITY, NEGATIVE\_INFINITY | Положительная и отрицательная бесконечность (бесконечно большие числа). |

Для получения значения указанных свойств не надо создавать объекта **Number**, а непосредственно обращаться к имени конструктора объекта (аналоги свойств класса в языках С++ и Java):

Number.NaN

Number.MAX\_VALUE

##### Методы объекта Number

|  |  |
| --- | --- |
| toExponential(точность) | Возвращает строку, представляющую экспоненциальную форму числа с заданным количеством цифр в дробной части. |
| toFixed(точность) | Возвращает строку, представляющую число в форме с фиксированной точкой с заданным количеством цифр в дробной части. |
| toPrecision(точность) | Возвращает строку, представляющую число в форме с фиксированной точкой с заданным количеством цифр мантиссе. |

Число 1760/7, хранящееся в переменной a, методом

a.toExponential(10) преобразуется в строку 2.5142857143e+2

a.toFixed(10) преобразуется в строку 251.4285714286

a.toPrecision(10) преобразуется в строку 251.4285714

Объект **Number** используется достаточно редко.

**3.3 Работа с формами, слоями, изображениями**

Контейнер FORM

Если рассматривать программирование на JavaScript в исторической перспективе, то первыми объектами, для которых были разработаны методы и свойства, стали поля форм. Обычно контейнер FORM и поля форм именованы:

<FORM NAME=fname METHOD=get>

<INPUT NAME=iname SIZE=30 MAXLENGTH=30>

</FORM>

Поэтому в программах на JavaScript к ним обращаются по имени:

document.fname.iname.value="Текст";

Того же эффекта можно достичь, используя коллекции форм и элементов, обращаясь к форме и к элементу либо по индексу, либо по имени:

document.forms[0].elements[0].value="Текст";

document.forms['fname'].elements['iname'].value="Текст";

Рассмотрим подробнее объект Form, который соответствует контейнеру FORM. Его свойства, методы и события используются для задания реакции на действия пользователя, например, изменения значений полей или нажатие кнопок.

|  |  |  |
| --- | --- | --- |
| **Свойства** | **Методы** | **События** |
| length | reset() | Reset |
| action | submit() | Submit |
| method |  |  |
| target |  |  |
| encoding |  |  |
| elements[] |  |  |

**Свойства объекта Form**

action

Свойство action отвечает за вызов CGI-скрипта. В нем указывается URL этого скрипта. Но там, где можно указать URL, можно указать и его схему javascript:, например:

<FORM METHOD=post ACTION="javascript: alert('Работает!');">

<INPUT TYPE=submit VALUE="Продемонстрировать JavaScript в ACTION">

</FORM>

В контейнере FORM указан атрибут METHOD. В данном случае это сделано для того, чтобы к URL, заданному в атрибуте ACTION, не дописывался символ "?". Дело в том, что методом доступа по умолчанию является метод GET. В этом методе при обращении к ресурсу из формы создается элемент URL под названием search. Этот элемент предваряется символом "?", который дописывается в конец URL скрипта. В данном случае это привело бы к неправильной работе JavaScript-кода, поскольку конструкция вида

alert('Строка');?

провоцирует ошибку JavaScript. Метод POST передает данные формы скрипту в теле HTTP-сообщения, поэтому символ "?" не добавляется к URL, и ошибка не генерируется. При этом применение void(0) отменяет перезагрузку документа, и браузер не генерирует событие Submit, т.е. не обращается к серверу при нажатии на кнопку, как это было бы при стандартной обработке формы.

method

Свойство method определяет метод доступа к ресурсам HTTP-сервера из программы-браузера. В зависимости от того, как автор HTML-страницы собирается получать и обрабатывать данные из формы, он может выбрать тот или иной метод доступа. На практике чаще всего используются методы GET и POST.

JavaScript-программа может изменить значение этого свойства. В предыдущем разделе метод доступа в форме был указан явно. Теперь мы его переопределим в момент исполнения программы:

<FORM NAME=f ACTION="javascript: alert(document.f.method);">

<INPUT TYPE=button onClick="document.f.method='post'" VALUE="Сменить метод на POST">

<INPUT TYPE=button onClick="document.f.method='get';" VALUE="Сменить метод на GET"><BR>

<INPUT TYPE=submit VALUE="JavaScript в ACTION">

</FORM>

target

Свойство target определяет имя окна, в которое следует загружать результат обращения к CGI-скрипту. При этом всегда есть альтернативы: можно использовать значение этого свойства внутри JavaScript-программ для указания окна или фрейма, куда требуется загружать результат работы CGI-скрипта, а можно получить идентификатор окна или задействовать встроенный массив frames[0] и свойства окна opener, top и parent.

encoding

Свойство encoding объекта Form (а также атрибут enctype контейнера FORM) задает, каким образом данные из формы должны быть закодированы перед их отправкой на сервер. Возможные значения:

|  |  |
| --- | --- |
| Значение | Описание |
| application/x-www-form-urlencoded | Это значение по умолчанию. Означает, что в данных, передаваемых на сервер, пробелы заменяются на "+", а специальные символы заменяются на их 16-ричное ASCII значение, например, буква Щ заменяется на %D0%A9. |
| text/plain | Пробелы заменяются на "+", но специальные символы не кодируются (передаются как есть). |
| multipart/form-data | Никакие символы не кодируются (они передаются как есть). Данное значение необходимо указывать, если в форме имеются элементы отправки файлов: <INPUT TYPE=file>. |

elements[]

При генерации встроенного в документ объекта Form браузер создает и связанный с ним массив (коллекцию) полей формы elements[]. Обычно к полям обращаются по имени, но можно обращаться и по индексу массива полей формы:

<FORM NAME=f>

<INPUT NAME=e SIZE=40>

<BR><INPUT TYPE=button VALUE="Ввести текст по имени элемента"

onClick="document.f.e.value='Текст введен по имени элемента';">

<BR><INPUT TYPE=button VALUE="Ввести текст по индексу элемента"

onClick="document.f.elements[0].value='Текст введен по индексу элемента';">

<BR><INPUT TYPE=reset VALUE="Очистить">

</FORM>

Индексирование полей в массиве начинается с нуля. Общее число полей в форме f доступно двумя способами: как свойство массива document.f.elements.length и как свойство объекта формы: document.f.length

**Методы объекта Form**

submit()

Метод submit() позволяет проинициировать передачу введенных в форму данных на сервер:

<FORM NAME=f ACTION="http://www.intuit.ru/rating\_students/">

Ваше имя пользователя на intuit:<INPUT NAME=query>

</FORM>

<A HREF="javascript:document.f.submit();">Посмотреть рейтинг</A>

В данном примере кнопки отправки (submit) у формы нет, но нажав на ссылку, выполняется отправка данных на сервер.

reset()

Метод reset() позволяет восстановить значения полей формы, заданные по умолчанию. Другими словами, вызов метода reset() равносилен нажатию на кнопку INPUT типа TYPE=reset, но при этом саму эту кнопку создавать не требуется.

<FORM NAME=f>

<INPUT VALUE="Значение по умолчанию" SIZE=30>

<INPUT TYPE=button VALUE="Изменим текст в поле ввода"

onClick="document.f.elements[0].value='Изменили текст';">

</FORM>

<A HREF="javascript:document.f.reset();void(0);">

Установили значение по умолчанию</A>

В данном примере если кликнуть по гипертекстовой ссылке, то в форме происходит восстановление значений полей по умолчанию.

**События объекта Form**

Submit

Событие Submit возникает (и соответствующий обработчик события onSubmit вызывается) при нажатии пользователем на кнопку типа submit или при выполнении метода submit(). Действие по умолчанию, которое выполняет браузер при возникновении этого события — отправка введенных в поля формы данных на сервер, указанный в атрибуте ACTION, с помощью метода, указанного в атрибуте METHOD, с использованием способа кодирования, указанного в арибуте ENCTYPE, и с указанием того, что результаты работы CGI-скрипта должны быть показаны в окне или фрейме с именем, указанным в атрибуте TARGET.

Фукцию обработки этого события можно переопределить и даже вовсе отменить. Для этой цели введен атрибут onSubmit="код\_программы" у контейнера <FORM>. В нем можно указать действия (JavaScript-код), какие должны выполняться при возникновении этого события.

Reset

Событие Reset возникает (и соответствующий обработчик события onReset вызывается) при нажатии пользователем на кнопку типа reset или при выполнении метода reset(). Действие по умолчанию, которое выполняет браузер при возникновении этого события — восстановление значений по умолчанию в полях формы. Однако функцию обработки этого события можно переопределить и даже вовсе отменить. Для этой цели введен атрибут onReset="код\_программы" у контейнера <FORM>. В нем можно указать действия (JavaScript-код), какие должны выполняться при возникновении этого события. Браузер сначала выполняет эти действия, а затем — свое действие по умолчанию. Но если последним оператором в обработчике onReset будет return false, то действие браузера по умолчанию выполняться не будет. Этот прием называется перехватом события. Пример:

<FORM onReset="javascript: alert('Не дадим восстановить!');return false;">

<INPUT VALUE="Измените этот текст" SIZE=30>

<INPUT TYPE=reset VALUE="Восстановить">

</FORM>

Здесь команда return false предотвратила восстановление значения поля. Команда return true, равно как и отсутствие оператора return, позволило бы браузеру продолжить обработку события — и восстановить значение поля.

**Поля формы и их объекты**

Как было сказано ранее, контейнеру <FORM> соответствует объект класса Form; он является свойством объекта document. В свою очередь, элементы формы, вложенные в контейнер <FORM>, например, <INPUT> различных типов, тоже соответствуют объектам различных классов, причем эти объекты являются свойствами объекта f.

У всех объектов, отвечающих полям формы, есть несколько стандартных свойств, доступных только для чтения: name (имя элемента, заданное в атрибуте NAME), type (тип элемента, например, для контейнеров <INPUT TYPE="..."> он совпадает со значением атрибута TYPE), form (указывает на форму f, в которой данный элемент содержится).

При программировании форм часто требуется писать обработчики событий для форм или их элементов, при этом нужно ссылаться на свойства данного элемента, других элементов и формы в целом.

document.форма.элемент.свойство // точечная нотация

document.форма.элемент["свойство"] // скобочная нотация

document.forms["имя\_формы"].elements["имя\_элемента"].свойство

document.forms[индекс\_формы].elements[индекс\_элемента].свойство

Однако получающиеся выражения — довольно громоздкие. Поэтому было введено следующее соглашение: в обработчике события формы или элемента формы имя текущего элемента можно опускать (вместе со всей предшествующей "приставкой"). Кроме того, ссылаться на сам текущий элемент можно с помощью ключевого слова this.

Например, предположим, что у нас есть форма:

<FORM NAME=f>

<INPUT TYPE=text NAME=e value="Текст" onFocus="">

<INPUT TYPE=button NAME=b value="Кнопка" onClick="">

</FORM>

Тогда вместо полной записи:

<INPUT TYPE=text NAME=e value="Текст" onFocus="alert(document.f.e.value)">

можно использовать краткую, опустив приставку "document.f.e", указывающую на текущий элемент:

<INPUT TYPE=text NAME=e value="Текст" onFocus="alert(value)">

Более того, в этом контексте эквивалентны следующие записи:

value // короче не бывает!

this.value // здесь this ссылается на элемент "e"

form.e.value // form есть свойство объекта "e" (равное "f")

this.form.e.value // комбинируем оба способа

document.f.e.value // почти полная запись

window.document.f.e.value // это самая полная запись

document.f.e.form.e.value // можно итерировать "form.e."

Текстовое поле ввода (объект Text)

Поля ввода (контейнер INPUT типа TYPE=text) являются одним из наиболее популярных объектов программирования на JavaScript. Это объясняется тем, что, помимо использования по прямому назначению, их применяют и в целях отладки программ, выводя в эти поля промежуточные значения переменных и свойств объектов.

<A HREF="http://site.com/">ссылка 1</A>

<FORM>Число гипертекстовых ссылок к данному моменту:

<SCRIPT>

document.write('<INPUT NAME=t VALUE='+document.links.length+'>');

</SCRIPT>

<BR><INPUT TYPE=button VALUE="Число ссылок по окончании загрузки страницы"

onClick="form.t.value=document.links.length;">

<BR><INPUT TYPE=reset>

</FORM>

<A HREF="http://rite.com/">ссылка 2</A>

В данном примере первое поле формы — это поле ввода. Мы присваиваем ему значение по умолчанию, равное числу гипертекстовых ссылок, имеющихся выше этого места в HTML-документе. Затем при помощи кнопки изменяем это значение на общее количество гипертекстовых ссылок во всем HTML-документе.

С каждым текстовым полем ввода <INPUT TYPE=text> связан свой объект класса Text, который является свойством той формы, в которой он был описан. Этот объект, в свою очередь, характеризуется следующими свойствами, методами и событиями:

|  |  |  |  |
| --- | --- | --- | --- |
| **Свойства** | **Методы** | **Обработчики событий** | |
| defaultValue | focus() | onChange | onMouseOver |
| value | blur() | onSelect | onMouseOut |
| size | select() | onChange | onMouseDown |
| maxLength |  | onSelect | onMouseUp |
| disabled |  | onClick | onKeyPress |
| readOnly |  | onDblClick | onKeyDown |
|  |  |  | onKeyUp |

Все перечисленные свойства можно менять: **value** (текущее значение поля ввода), **defaultValue** (значение поля ввода по умолчанию), **size** (число умещающихся в поле символов, т.е. видимых) **maxLength** (максимальное число символов, которое можно присвоить значению данного поля) **readOnly** (может ли пользователь менять значение поля) **disabled** (может ли пользователь установить фокус на этом поле).

Методы: **focus**() — устанавливает фокус на данном поле, **blur**() — убирает фокус с данного поля, **select**() — выделяет весь введенный текст (чтобы, например, его можно было скопировать в буфер, либо удалить, нажав клавишу Delete).

Списки вариантов (объекты **Select** и **Option**)

Одним из важных элементов интерфейса пользователя являются списки вариантов. В HTML-формах для их реализации используется контейнер <SELECT>, который вмещает в себя контейнеры <OPTION>. При этом список может "выпадать" либо прокручиваться внутри окна. В зависимости от наличия атрибута MULTIPLE у контейнера <SELECT> список может быть либо с возможностью выбора только одного варианта, либо нескольких вариантов.

С каждым контейнером <SELECT> ассоциирован объект класса Select, а с каждым дочерним контейнером <OPTION> — объект класса Option, являющийся свойством данного объекта класса Select. Кроме того, свойством объекта класса Select является также коллекция options[], объединяющая все его дочерние объекты Option. Перечислим основные свойства, методы и события, характеризующие эти объекты.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Объект **Select** | | | Объект **Option** | |
| **Свойства** | **Методы** | **Обработчики событий** | **Свойства** | **Методы и событиия** |
| options[] | focus() | onBlur | defaultSelected | нет |
| size | blur() | onChange | selected |
| length |  | onFocus | index |
| multiple | add() |  | text |
| selectedIndex | remove() |  | value |

Создание объектов **Option**

Объект класса Option имеет конструктор, с помощью которого можно создать объект класса Option:

opt = new Option([ text, [ value, [ defaultSelected, [ selected ]]]]);

где аргументы соответствуют свойствам обычных объектов класса Option:

text — строка текста, которая размещается в контейнере <OPTION> (например: <OPTION>текст</OPTION>);

value — значение, которое передается серверу при выборе альтернативы, связанной с объектом Option;

defaultSelected — выбрана ли эта альтернатива по умолчанию (true/false);

selected — альтернатива была выбрана пользователем (true/false).

У объектов класса Option нет свойства name, в виду того, что у контейнера <OPTION> нет атрибута NAME. Таким образом, к встроенным в документ объектам класса Option можно обращаться только как к элементам коллекции options[].

Коллекция **options**[]

Встроенный массив (коллекция) options[] — это одно из свойств объекта Select. Элементы этого массива являются полноценными объектами класса Option. Они создаются по мере загрузки страницы браузером. Количество объектов Option, содержащихся в объекте document.f.s класса Select, можно узнать с помощью стандартного свойства массива: document.f.s.options.length. Кроме того, у самого объекта Select есть такое же свойство: document.f.s.length — оно полностью идентично предыдущему.

Можно не только создавать новые объекты Option, но и удалять уже созданные браузером объекты:

<FORM>

<SELECT NAME=s>

<OPTION>Первый вариант</OPTION>

<OPTION>Второй вариант</OPTION>

<OPTION>Третий вариант</OPTION>

</SELECT>

<INPUT TYPE=button VALUE="Удалить последний вариант"

onClick="form.s.options[form.s.length-1]=null;">

<INPUT TYPE=reset VALUE="Сбросить">

</FORM>

В данном примере при загрузке страницы с сервера имеются три альтернативы. Их можно просматривать как ниспадающий список вариантов. После нажатия на кнопку "Удалить последний вариант" в форме остается только две альтернативы. Если еще раз нажать на эту кнопку, останется только одна альтернатива. В конечном счете, вариантов не останется вовсе, т.е. пользователь лишится возможности выбора. При нажатии кнопки сброса (reset) варианты не восстанавливаются — альтернативы утеряны бесследно.

При использовании конструктора Option, можно сделать процесс обратимым:

<SCRIPT>

function RestoreOptions()

{

document.f.s.options[0] = new Option('Первый вариант','',true,true);

document.f.s.options[1] = new Option('Второй вариант');

document.f.s.options[2] = new Option('Третий вариант');

return false;

}

</SCRIPT>

<FORM NAME=f onReset="RestoreOptions();">

<SELECT NAME=s>

<OPTION>Первый вариант</OPTION>

<OPTION>Второй вариант</OPTION>

<OPTION>Третий вариант</OPTION>

</SELECT>

<INPUT TYPE=button VALUE="Удалить последний вариант"

onClick="form.s.options[form.s.length-1]=null;">

<INPUT TYPE=reset VALUE=Reset>

</FORM>

Восстановление альтернатив поместили в обработчик события onReset контейнера FORM. Создаваемые объекты класса Option подчиняем объекту document.f.s класса Select. При этом первая альтернатива должна быть выбранной по умолчанию (аргументу defaultSelected задано значение true), чтобы смоделировать поведение при начальной загрузке страницы.

Свойства **text** и **value** объекта Option

Свойство text представляет собой отображаемый в меню текст, который соответствует альтернативе. В HTML-коде он расположен между тэгами <OPTION> и </OPTION>. Свойство value содержит значение атрибута VALUE тэга <OPTION>. Например, пусть один из вариантов в списке был описан как:

<OPTION VALUE="n1">Вариант первый</OPTION>

Тогда значение свойства text у соответствующего объекта будет равно "Вариант первый", а значение свойства value равно "n1".

Зачем нужны два свойства? Дело в том, что на сервер передается значение value выбранного варианта. В случае же, когда атрибут VALUE у контейнера <OPTION> отсутствует, на сервер передается значение text.

Свойства **selected** и **selectedIndex**

Свойство selectedIndex объекта Select возвращает номер выбранного варианта (нумерация начинается с нуля).

<FORM> Вариант:

<SELECT onChange="form.e.value=selectedIndex;">

<OPTION>Один</OPTION>

<OPTION>Два</OPTION>

</SELECT>

Выбрали индекс: <INPUT NAME=e>

</FORM>

В обработчике события onChange ссылаемся на второй элемент формы. На данный момент он не определен, но событие произойдет только тогда, когда будем выбирать вариант — к этому моменту поле уже будет определено.

Если список вариантов задан как <SELECT MULTIPLE>, т.е. с возможностью выбора нескольких опций одновременно, то свойство selectedIndex возвратит индекс первой выбранной опции. На этот случай имеется альтернатива: свойство selected у каждого объекта Option. Оно равно true, если данная опция выбрана, и false в противном случае.

Обработчик события **onChange** объекта **Select**

Событие Change наступает в тот момент, когда пользователь меняет свой выбор вариантов. Если поле является полем выбора единственного варианта, то все просто — см. предыдущий пример. Посмотрим, что происходит, когда имеем дело с полем выбора множественных вариантов:

<FORM>

Фрукты: <SELECT MULTIPLE

onChange="form.e.value='';

for(i=0; i<length; i++)

if(options[i].selected)

form.e.value += options[i].text+', ';">

<OPTION>яблоко</OPTION>

<OPTION>банан</OPTION>

<OPTION>киви</OPTION>

<OPTION>персик</OPTION>

</SELECT><BR>

Выбраны позиции: <INPUT READONLY SIZE=70 NAME=e>

</FORM>

Событие Change происходит тогда, когда пользователь выбирает или отменяет какой-либо вариант. Исключение составляет тот случай, когда варианты при выборе последовательно отмечаются (нажатие кнопки мыши на одном элементе, ведение мыши до конечного элемента, отпускание кнопки мыши). В этом случае событие происходит в тот момент, когда пользователь отпускает кнопку мыши, и все отмеченные альтернативы становятся выбранными.

Кнопки

В HTML-формах используется четыре вида кнопок:

<FORM>

<INPUT TYPE=button VALUE="Кнопка типа button">

<INPUT TYPE=submit VALUE="Кнопка отправки">

<INPUT TYPE=reset VALUE="Кнопка сброса">

<INPUT TYPE=image SRC=a.gif> <!-- графическая кнопка -->

</FORM>

В атрибуте кнопки можно задать обработчик события onClick, а в атрибуте формы — обработчики событий onSubmit и onReset. Кроме того, кнопкам и форме соответствуют объекты DOM. Объект, отвечающий кнопке, имеет метод click(). Объект, отвечающий форме, имеет методы submit() и reset(). При программном вызове метода submit() нужно позаботиться о дополнительном вызове обработчика события onSubmit, чтобы, например, данные не были отправлены на сервер без предварительной проверки.

Кнопка **button**

Кнопка типа button вводится в форму главным образом для того, чтобы можно было выполнить какие-либо действия либо при ее нажатии пользователем, либо при вызове метода click().

<FORM NAME=f>

<INPUT TYPE=button NAME=b VALUE="Кнопка" onClick="alert('5+7='+(5+7))">

</FORM>

<A HREF="javascript:document.f.b.click();void(0);">Вызвать метод click()</A>

Кнопка **submit**

Кнопка отправки (submit) позволяет отправить данные, введенные в форму, на сервер. В простейшем случае — при отсутствии у контейнера <FORM> атрибутов ACTION (его значением по умолчанию является адрес текущей страницы), METHOD (его значением по умолчанию является GET) и TARGET (его значением по умолчанию является \_self) — стандартным действием браузера при отправке данных на сервер является просто перезагрузка текущей страницы, что подтверждает следующий пример:

<FORM>

<INPUT TYPE=submit>

</FORM>

Для имитации ответа сервера заготовим следующий простой HTML-файл **receive.htm**:

<HTML><BODY>Данные приняты!</BODY></HTML>

Усложним пример: добавим обработчики событий onClick (у кнопки отправки) и onSubmit (у формы)

<FORM NAME=f ACTION="receive.htm"

onSubmit="return confirm('Вы хотите отправить данные?')">

<INPUT onClick="alert('Вызван обработчик onClick у кнопки отправки')"

TYPE=submit VALUE="Кнопка отправки" NAME=s>

</FORM>

Соответственно, для выполнения дополнительных действий перед отправкой данных можно поместить код в любой из указанных обработчиков; в частности, поместив в какой-либо из них оператор return false, можно предотвратить отправку данных.

Вызов метода click() кнопки отправки равносилен нажатию этой кнопки — произойдут все три вышеперечисленных действия:

<FORM NAME=f ACTION="receive.htm"

onSubmit="return confirm('Вы хотите отправить данные?')">

<INPUT onClick="alert('Вызван обработчик onClick у кнопки отправки')"

TYPE=submit VALUE="Кнопка отправки" NAME=s></FORM>

<A HREF="javascript: document.f.s.click();void(0);" >Вызвать метод <B>click()</B> кнопки отправки</A>

Метод **submit**() формы

Вызов метода submit() формы не равносилен нажатию кнопки отправки. При вызове этого метода будет выполнена отправка данных на сервер.

<FORM NAME=f ACTION="receive.htm"

onSubmit="return confirm('Вы хотите отправить данные?')">

<INPUT onClick="alert('Вызван обработчик onClick у кнопки отправки')"

TYPE=submit VALUE="Кнопка отправки" NAME=s></FORM>

<A HREF="javascript: document.f.submit();void(0);"

>Вызвать метод <B>submit()</B> формы</A>

Тем самым данные могут уйти на сервер без предварительной проверки JavaScript-скриптом. Каким образом заставить браузер вызвать обработчик onSubmit? Для этого существует возможность обратиться к этому обработчику напрямую: document.f.onsubmit() Остается предусмотреть, что после этого метод submit() должен вызываться не всегда, а только если onSubmit либо не возвратил никакого значения, либо возвратил true, иными словами, если он не возвратил false. Окончательно получаем:

<FORM NAME=f ACTION="receive.htm"

onSubmit="return confirm('Вы хотите отправить данные?')">

<INPUT onClick="alert('Вызван обработчик onClick у кнопки отправки')"

TYPE=submit VALUE="Кнопка отправки" NAME=s></FORM>

<A HREF="javascript:

if(document.f.onsubmit() != false)

document.f.submit(); void(0);"

>Вызвать <B>submit()</B> с преварительной проверкой onSubmit</A>

Есть еще один способ инициировать отправку данных формы в обход кнопки отправки (которой, кстати, у формы может и не быть). Если фокус находится на любом текстовом поле <INPUT TYPE=text> формы и пользователь нажмет клавишу **Enter**, то (в большинстве браузеров) произойдет вызов обработчика события onSubmit формы и отправка данных на сервер.

Введите текст и нажмите Enter:<BR>

<FORM ACTION="receive.htm"

onSubmit="return confirm('Вы хотите отправить данные?')">

<INPUT TYPE=text VALUE="Текст вводить здесь:" SIZE=50>

</FORM>

Этот способ работает логичнее, чем метод submit(), т.к. отправляемые на сервер данные не избегают предварительной проверки обработчиком onSubmit.

Кнопка **reset**

Кнопка сброса (reset) позволяет вернуть все поля формы в первоначальное состояние, которое они имели при загрузке страницы.

Вызов метода click() у кнопки сброса равносилен нажатию этой кнопки, т.е. приводит к тем же действиям:

<FORM NAME=f onReset="return confirm('Вы хотите очистить форму?')">

<INPUT TYPE=text VALUE="Измените этот текст">

<INPUT TYPE=reset VALUE="Кнопка сброса" NAME=s onClick="alert('Вызван обработчик onClick у кнопки сброса')">

</FORM>

<A HREF="javascript: document.f.s.click();void(0);">Вызвать метод <B>click()</B> кнопки сброса</A>

Есть способы сбросить форму в исходное состояние в обход кнопки сброса (которой, кстати, у формы может и не быть). Во-первых, это вызов метода reset() у формы. Во-вторых, если фокус находится на любом поле или кнопке формы, то можно нажать клавишу **Esc**. Пример:

Измените текст, а затем нажмите Esc (либо ссылку).<BR>

<FORM NAME=f onReset="return confirm('Вы хотите очистить форму?')">

<INPUT TYPE=text VALUE="Измените этот текст">

</FORM>

<A HREF="javascript: document.f.reset();void(0);">Вызвать метод<B>reset()</B> формы</A>

Как можно видеть, оба способа не просто сбрасывают форму, но и вызывают обработчик события onReset формы. Таким образом, метод reset() ведет себя более логично и предсказуемо, нежели submit().

Графическая кнопка

Графическая кнопка — это разновидность кнопки отправки. Ее отличие в том, что вместо кнопки с надписью пользователь увидит картинку, по которой можно кликнуть:

<FORM ACTION="receive.htm">

<INPUT TYPE=image SRC="pic.gif">

</FORM>

Кроме того, когда пользователь кликает по графической кнопке, то на сервер отправятся не только данные, введенные в поля формы, но также и координаты указателя мыши относительно левого верхнего угла изображения. К сожалению, перехватить эти координаты в JavaScript-программе не удается. Если Вам необходимо работать с этими координатами, то вместо графической кнопки рекомендуется создать активную карту с помощью контейнера <MAP>.

Графические кнопки имеют ряд странностей. Например, являясь одновременно и кнопкой, и изображением, они почему-то отсутствуют как в коллекции **document.f.elements[]**, так и в коллекции **document**.**images[]** (IE 7, Mozilla Firefox). Как следствие, они не учитываются ни в общем количестве элементов формы (**document.f.length**), ни в общем количестве изображений документа (**document.images.length**).

Обратиться к такой кнопке можно задав атрибут ID:

<INPUT TYPE=image SRC="pic.gif" ID="d1">

и затем в программе написав:

var knopka = document.getElementById('d1')

После этого мы можем обращаться к свойствам этой кнопки, например **knopka.src**, а также к методу **knopka.click()** .

Следующий пример показывает, что вызов метода click() графической кнопки "почти" равносилен нажатию этой кнопки, т.е. последовательно вызывает обработчики onClick кнопки, onSubmit формы и передает данные на сервер (но что при этом передается в качестве координат курсора мыши):

<FORM ACTION="receive.htm"

onSubmit="return confirm('Вы хотите отправить данные?')">

<INPUT onClick="alert('Вызван обработчик onClick у графической кнопки')"

TYPE="image" SRC="pic.gif" id="d1">

</FORM>

<A HREF="javascript:var knopka = document.getElementById('d1');knopka.click();void(0);"> Вызвать метод <B>click()</B> графической кнопки</A>

**Что такое слои?**

Слои - это одна из замечательных новых возможностей браузера Netscape Navigator 4.0. Она позволяет выполнять точное позиционирование таких объектов web-страницы, как изображения. Кроме того, теперь Вы можете перемещать объекты по вашей HTML-странице. Вы можете также делать объекты невидимыми. Управлять слоями можно легко с помощью языка JavaScript. Я надеюсь, что Вы, как и я, проникнитесть энтузиазмом в отношении возможностей этих слоев.

В данный момент пользоваться слоями Вы можете только в Netscape Navigator 4.0!

И, как обычно, я не буду детально описывать различные тэги. Хороший документ, описывающий все свойства такого объекта Netscape Navigator 4.0, как слои, находится по адресу http://home.netscape.com/comprod/products/communicator/index.html - и потому у меня нет нужды его пересказывать.

Что такое в действительности слои? Объяснять это очень легко на простом примере: берете несколько листов бумаги. На одном листе пишете текст. На другом - рисуете картинку. На третьем листе берете картинку и вписываете вокруг нее некий текст. И так далее. Теперь кладете эти листы на стол. Допустим, каждый лист - это некий слой. В этом смысле слой представляет собой в некотором роде контейнер. То есть он может включать в себя (содержать) некие объекты - в данном случае это будет текст и изображения.   
Теперь берем бумагу с изображением и двигаем его по столу. Внимательно следите за тем, как это изображение движется вслед за бумагой. Если Вы сместим лист бумаги вправо, то и изображение тоже сдвинется! И что же собственно мы должны извлечь из этого увлекательного опыта? Слои, способные содержать различные объекты, например изображения, формы, текст, могут быть наложены на Вашу HTML-страницу и даже перемещаться по ней. Если Вы сдвигаете какой-либо слой, то и все содержащиеся в этом слое объекты тоже будут повторять это движение.   
Слои могут накладываться друг на друга подобно простым листам бумаги на столе. В каждом слое могут присутствовать прозрачные области. Сделайте в листе бумаги дырку. Теперь положите этот лист на другой. Такое отверстие - это 'прозрачная область' на первом листе - и через нее как раз видны нижележащие листы.

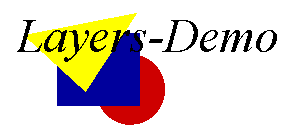
**Создание слоев**

Чтобы создать слой, мы должны использовать либо тэг <layer> либо <ilayer>. Вы можете воспользоваться следующими параметрами:

|  |  |
| --- | --- |
| name="*layerName*" | Название слоя |
| left=*xPosition* | Абсцисса левого верхнего угла |
| top=*yPosition* | Ордината левого верхнего угла |
| z-index=*layerIndex* | Номер индекса для слоя |
| width=*layerWidth* | Ширина слоя в пикселах |
| clip="*x1\_offset, y1\_offset, x2\_offset, y2\_offset"* | Задает видимую область слоя |
| above="*layerName*" | Определяет, какой слой окажется под нашим |
| below="*layerName*" | Определяется, какой слой окажется над нашим |
| Visibility=show|hide|inherit | Видимость этого слоя |
| bgcolor="*rgbColor*" | Цвет фона - либо название стандартного цвета, либо rgb-запись |
| background="*imageURL*" | Фоновая картинка |

Тэг <layer> используется для тех слоев, которые Вы можете точно позиционировать. Если же Вы не указываете положение слоя (с помощью параметров *left* и *top*), то по умолчанию он помещается в верхний левый угол окна.   
Тэг <ilayer> создает слой, положение которого определяется при формировании документа.

Давайте теперь начнем с простого примера. Мы хотим создать два слоя. В первом из них мы помещаем изображение, а во втором - текст. Все, что мы хотим сделать - показать этот текст поверх данного изображения.

  
Текст поверх изображения

Исходный код:

<html>

<layer name=pic z-index=0 left=200 top=100>

<img src="img.gif" width=160 height=120>

</layer>

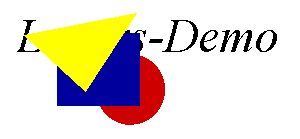
<layer name=txt z-index=1 left=200 top=100>

<font size=+4> <i> Layers-Demo </i> </font>

</layer>

</html>

Как видим, с помощью тэга <layer> мы формируем два слоя. Оба слоя позиционируются как 200/100 (через параметры *left* и *top*). Все, что находится между тэгами <layer> и </layer> (или тэгами <ilayer> и </ilayer>) принадлежит описываемому слою.   
Кроме того, мы используем параметр *z-index*, определяя тем самым порядок появления указанных слоев - то есть, в нашем случае, Вы тем самым сообщаете браузеру, что текст будет написан поверх изображения. В общем случае, именно слой с самым высоким номером z-index будет показан поверх всех остальных. Вы не ограничены в выборе z-index лишь значениями 0 и 1 - можно выбирать вообще любое положительное число.   
Так, если в первом тэге <layer> Вы напишете *z-index=100*, то текст окажется под изображением - его слой номер Z-индекса (z-index=1). Вы сможете увидеть текст сквозь изображение, поскольку я использовал в нем прозрачный фон (формат gif89a).

  
Тект под изображением

**Слои и JavaScript**

Рассмотрим теперь, как можно получить доступ к слоям через JavaScript. Начнем же мы с примера, где пользователь получает возможность, нажимая кнопку, прятать или показать некий слой.   
Для начала мы должны знать, каким образом слои представлены в JavaScript. Как обычно, для этого имеются несколько способов. Самое лучшее - дать каждому слою свое имя. Так, если мы задаем слой

<layer ... name=myLayer>

</layer>

то в дальнейшем можем получить доступ к нему с помощью конструкции*document.layers["myLayer"]*. Согласно документации, предоставляемой фирмой Netscape, мы можем также использовать запись *document.myLayer* - однако в моем браузере это приводит к сбою. Конечно, это всего лишь проблема предварительной версии и в заключительном варианте будет успешно решена (сейчас я пользуюсь Netscape Navigator 4.0 PR3 на WinNT). Однако, по-видимому, нет никаких проблем с конструкцией *document.layers["myLayer"]* - поэтому мы и будем пользоваться именно такой альтернативой из всех возможных.   
Доступ к этим слоям можно также получить через целочисленный индекс. Так, чтобы получить доступ к самому нижнему слою, Вы можете написать *document.layers[0]*. Обратите внимание, что индекс - это не то же самое, что параметр z-index. Если, например, Вы имеете два слоя, называемые *layer1* и *layer2* с номерами z-index 17 и 100, то Вы можете получить доступ к этим слоям через *document.layers[0]* и *document.layers[1]*, а **не** через *document.layers[17]* и *document.layers[100]*.

Слои имеют несколько свойств, которые можно изменять с помощью скрипта на JavaScript. В следующем примере представлена кнопка, которая позволяет Вам скрывать или, наоборот, предоставлять один слой (требуется Netscape Navigator версии 4.0 или выше).

*This text is inside a layer*

Исходный код скрипта выглядит следующим образом:

<html>

<head>

<script language="JavaScript">

<!—hide

function showHide() {

if (document.layers["myLayer"].visibility == "show")

document.layers["myLayer"].visibility= "hide"

else document.layers["myLayer"].visibility= "show";

}

// -->

</script>

</head>

<body>

<ilayer name=myLayer visibility=show>

<font size=+1 color="#0000ff"><i>This text is inside a layer</i></font>

</ilayer>

<form><input type="button" value="Show/Hide layer" onClick="showHide()">

</form>

</body>

</html>

Данная кнопка вызывает функцию *showHide()*. Можно видеть, что в этих функциях реализуется доступ к такому свойству объекта layer (*myLayer*), как *видимость*. Присвоивая параметру *document.layers["myLayer"].visibility* значения *"show"* или *"hide"*, Вы можете показать или скрыть наш слой. Заметим, что *"show"* и *"hide"* - это строки, а не зарезервированные ключевые слова, то есть Вы **не можете** написать*document.layers["myLayer"].visibility= show*.   
Вместо тэга <layer> я также пользовался тэгом <ilayer>, поскольку хотел поместить этот слой в "информационный поток" документа.

**Перемещение слоев**

Свойства *left* и *top* определяют задают положение данного слоя. Вы можете менять его, записывая в эти атрибуты новые значения. Например, в следующей строке задается горизонтальное положение слоя в 200 пикселов:

document.layers["myLayer2"].left= 200;

Перейдем теперь к программе перемещения слоев - она создает нечто вроде прокрутки внутри окна браузера.

*This text is inside a layer*

Сам скрипт выглядит следующим образом:

<html>

<head>

<script language="JavaScript">

<!—hide

function move() {

if (pos < 0) direction= true;

if (pos > 200) direction= false;

if (direction) pos++

else pos--;

document.layers["myLayer2"].left= pos;

}// -->

</script>

</head>

<body onLoad="setInterval('move()', 20)">

<ilayer name=myLayer2 left=0>

<font size=+1 color="#0000ff"><i>This text is inside a layer</i></font>

</ilayer>

</body>

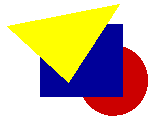
</html>

Мы создаем слой с именем *myLayer2*. Можно видеть, что в тэге <body>мы пользуемся процедурой *onLoad*. Нам необходимо начать прокручивание слоя, как только страница будет загружена. В процедуре обработки события *onLoad* мы пользуемся функцией *setInterval()*. Это один из новых методов версии 1.2 языка JavaScript (то есть версии JavaScript, реализованной в Netscape Navigator 4.0). Им можно пользоваться, чтобы вызывать некую функцию мвновь и вновь через определенные интервалы времени. В прошлом для этого мы пользовались функцией *setTimeout()*. Функция *setInterval()* работает почти так же, однако Вам нужно вызвать ее всего лишь один раз.   
С помощью *setInterval()* мы вызываем функцию *move()* каждые 20 миллисекунд. А функция *move()*, в свою очередь, всякий раз смещает слой на новую позицию. И поскольку мы вызываем эту функцию вновь и вновь, то мы получаем быстрый скроллинг нашего текста. Все, что мы нужно сделать в функции *move()* - это вычислить новую координату для слоя и записать ее: *document.layers["myLayer2"].left= pos*.

Если Вы посмотрите исходный код этой части в онлайновом описании, то увидите, что в действительности мой код выглядит несколько иначе - я добавил некий фрагмент кода с тем, чтобы люди, работающие со старыми версиями JavaScript-браузеров, не получали из-за этого никаких сообщений об ошибках. Как этого можно достичь? Следующий фрагмент кода будет выполняться только на тех браузерах, которые воспринимают язык JavaScript 1.2:

<script language="JavaScript1.2">  
<!-- hide  
document.write("Вы используете браузер, совместимый с JavaScript 1.2.");  
// -->  
</script>

Та же самая проблема возникает, когда мы работаем с объектом Image. Мы можем аналогичным способом переписать кодекс. Установка переменной *browserOK* решает эту проблему. Следующий пример демонстрирует, как может осуществляться перекрывание слоев:

*This text is inside a layer* 

Мы уже обсудили основные понятия новой технологии слоев. В этой же части будут рассмотрены следующие темы:

* Вырезка из слоя
* Вложенные слои
* Различные эффекты с прозрачными слоями

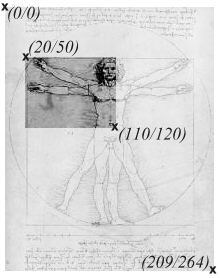
### Вырезка из слоя

Можно постулировать, что какая-то (прямоугольная) часть слоя будет нам видима. Все же, что лежит за ее пределами, показано на экране не будет. Такой прием называется вырезанием. Например, в разметке HTML можно задать следующую функцию *вырезания*:

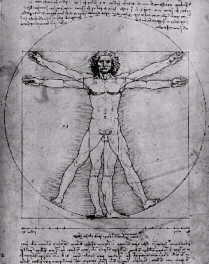
<ilayer left=0 top=0 clip="20,50,110,120">  
<img src="davinci.jpg" width=209 height=264>  
</ilayer>

(Здесь я приписал параметры *left=0* и *top=0*, поскольку в противном случае, если этого не сделать, то с моей версией Netscape (PR3 on WinNT) возникают некоторые проблемы)   
Хотя само изображение и имеет размеры 209x264 пикселов, мы можем видеть лишь его малую часть:

Данный фрагмент изображения имеет размер 90x70 (пикселов). Первые два значения, указанные в атрибуте *clip* (атрибуте HTML-тэга <layer> или <ilayer>), указывают верхний левый угол вырезаемой части. Следующие два значения указывают нижний правый угол. Сказанное можно проиллюстрировать следующим рисунком:



Еще более интересных результатов можно добиться, управляя вырезанной частью с помощью языка JavaScript. Точнее, Вы можете изменять значения свойств *clip.left*, *clip.top*, *clip.right* и *clip.bottom* объекта Layer. Достаточно всего лишь занести в одно из этих свойств новое значение, как фрагмент тут же будет кадрирован соответствующим образом . В следующем примере параметры вырезанной части изображения меняются динамически, и в результате у пользователя создается впечатление, будто изображение медленно "растет":



Код соответсвующего скрипта:

<html>  
<head>  
  
<script language="JavaScript">  
<!-- hide  
  
var middleX, middleY, pos;

function start() {  
 // получить размер изображени\я   
 var width= document.layers["imgLayer"].document.davinci.width;  
 var height= document.layers["imgLayer"].document.davinci.height;  
  
 // определить, какой пиксел находитс\я в центре изображени\я   
 middleX= Math.round(width/2);  
 middleY= Math.round(height/2);  
  
 // начальная позици\я   
 pos= 0;  
  
 // запуск!  
 show();  
}  
  
function show() {  
  
 // увеличить размер вырезаемой области   
 pos+= 2; // step size  
 document.layers["imgLayer"].clip.left= middleX- pos;  
 document.layers["imgLayer"].clip.top= middleY- pos;  
 document.layers["imgLayer"].clip.right= middleX+ pos;  
 document.layers["imgLayer"].clip.bottom= middleY+ pos;  
  
 // проверить, не высвечено ли все изображение   
 if (!((pos > middleX) && (pos > middleY)))   
 setTimeout("show()", 20);   
  
}  
  
// -->  
</script>  
</head>  
  
<body>  
  
<ilayer name="imgLayer" clip="0,0,0,0">  
<img name=davinci src="davinci.jpg" width=209 height=264>  
</ilayer>  
  
<form>  
<input type=button value="Start" onClick="start()">  
</form>  
  
</body>  
</html>

Кнопка, представленная в разделе *<body>*, вызывает функцию *start()*. Сначала мы должны определить точку, с которой нам следует начать работу - фактически это будет некий пиксел в центре нашего изображения. Значения координат x и y этого пиксела мы помещаем в переменные *middleX* и *middleY*. После этого мы вызываем функцию *show()*, которая задает размеры вырезаемой части изображения в зависимости от значений переменных *middleX*, *middleY* и параметра *pos*. При этом значение переменной *pos*автоматически увеличивается при каждом вызове функции *show()*. То есть размер вырезаемой части изображения с каждым разом становится все больше и больше. В самом конце процедуры *show()* мы устанавливаем таймер с помощью вызова *setTimeout()* - и благодаря этому функция *show()* вызывается вновь и вновь. И этот процесс остановится только тогда, когда изображение будет показано целиком.   
Заметим, что размер изображения мы получаем в самом начале функции *start()*:

var width= document.layers["imgLayer"].document.davinci.width;  
 var height= document.layers["imgLayer"].document.davinci.height;

С помощью конструкции *document.layers["imgLayer"]* мы можем обратиться к слою с именем *imgLayer*. Однако почему после *document.layers["imgLayer"]* мы ставим *document*? Дело в том, что каждый слой имеет свою собственную HTML-страницу - то есть, **каждый слой имеет свой объект document**. Чтобы получить доступ к изображению внутри слоя *imgLayer*, нам необходимо получить доступ к этому объекту document. В приведенном выше примере такое изображение носило название *davinci*. Все остальное поле листа должно быть чистым.

### Вложенные слои

Как мы уже видели, слой может содержать несколько различных объектов. Он могут даже включать в себя другие слои. Конечно, может возникнуть вопрос, для чего это нужно. На самом деле есть несколько причин, чтобы пользоваться вложенными слоями. Рассмотрим несколько примеров, демонстрирующих применение вложенных слоев.   
В первом примере используется слой (называемый *parentLayer*), в который вложено еще два других слоя (*layer1* и *layer2*).

Это первый слой

Это второй слой

Это главный (родительский) слой

После открытия страницы мы видим три кнопки. Эти кнопки могут запускать и останавливать движение слоев. Также можно видеть, что перемещение слоя *parentLayer*сопровождается перемещением и двух других слоев, тогда как перемещение слоя *layer1* (или *layer2*) ни на что другое не влияет. Этот пример демонстрирует возможность объединения группы объектов с помощью механизма вложенных слоев.

Рассмотрим теперь исходный код скрипта:

<html>  
<head>  
  
<script language="JavaScript">  
<!-- hide  
  
// начальна\я позици\я   
var pos0= 0;   
var pos1= -10;  
var pos2= -10;  
  
// движение?  
var move0= true;  
var move1= false;  
var move2= false;  
  
// направление?  
var dir0= false;  
var dir1= false;  
var dir2= true;  
  
function startStop(which) {  
 if (which == 0) move0= !move0;  
 if (which == 1) move1= !move1;  
 if (which == 2) move2= !move2;  
}  
  
function move() {  
  
 if (move0) {  
 // перемещение parentLayer   
 if (dir0) pos0--  
 else pos0++;  
  
 if (pos0 < -100) dir0= false;  
  
 if (pos0 > 100) dir0= true;  
  
 document.layers["parentLayer"].left= 100 + pos0;   
 }  
  
 if (move1) {  
 // перемещение parentLayer   
 if (dir1) pos1--  
 else pos1++;  
  
 if (pos1 < -20) dir1= false;  
  
 if (pos1 > 20) dir1= true;  
  
 document.layers["parentLayer"].layers["layer1"].top= 10 + pos1;  
 }  
  
 if (move2) {  
 // перемещение parentLayer   
 if (dir2) pos2--  
 else pos2++;  
  
 if (pos2 < -20) dir2= false;  
  
 if (pos2 > 20) dir2= true;  
  
 document.layers["parentLayer"].layers["layer2"].top= 10 + pos2;   
 }  
  
}  
  
// -->  
</script>  
</head>  
  
<body onLoad="setInterval('move()', 20)">  
  
<ilayer name=parentLayer left=100 top=0>  
 <layer name=layer1 z-index=10 left=0 top=-10>  
 Это первый слой   
 </layer>  
  
 <layer name=layer2 z-index=20 left=200 top=-10>  
 Это второй слой   
 </layer>  
  
 <br><br>  
 Это главный (родительский) слой   
  
</ilayer>  
  
<form>  
<input type="button" value="Move/Stop parentLayer" onClick="startStop(0);">  
<input type="button" value="Move/Stop layer1" onClick="startStop(1);">  
<input type="button" value="Move/Stop layer2" onClick="startStop(2);">  
</form>  
  
</body>  
</html>

Можно видеть, что внутри *parentLayer* мы определили два слоя. Это как раз и есть вложенные слои. Как получить к этим слоям доступ в языке JavaScript? Как это делается, можно посмотреть в функции *move()*:

document.layers["parentLayer"].left= 100 + pos0;  
 ...   
 document.layers["parentLayer"].layers["layer1"].top= 10 + pos1;  
 ...  
 document.layers["parentLayer"].layers["layer2"].top= 10 + pos2;

Чтобы получить доступ к вложенным слоям, Вам недостаточно будет просто написать *document.layers["layer1"]* или *document.layers["layer2"]*, поскольку слои *layer1* и *layer2* лежат **внутри** *parentLayer*.

Посмотрим теперь, как можно задать выделяемую область. В следующем примере используется механизм вырезания и перемещение изображения. Чего этим мы хотим достичь - чтобы вырезаемая часть была зафиксирована, т.е. чтобы при перемещении всего изображения не происходила смена видимого на экране фрагмента.

Исходный код скрипта:

<html>  
<head>  
  
<script language="JavaScript">  
<!-- hide  
  
var pos= 0; // начальное положение   
var direction= false;  
  
function moveNclip() {  
  
 if (pos<-180) direction= true;  
 if (pos>40) direction= false;  
  
 if (direction) pos+= 2  
 else pos-= 2;  
  
 document.layers["clippingLayer"].layers["imgLayer"].top= 100 + pos;  
  
}  
  
// -->  
</script>  
  
</head>  
<body onLoad="setInterval('moveNclip()', 20);">  
  
<ilayer name="clippingLayer" z-index=0 clip="20,100,200,160" top=0 left=0>  
 <ilayer name="imgLayer" top=0 left=0>  
 <img name=davinci src="davinci.jpg" width=209 height=264>  
 </ilayer>  
</ilayer>  
  
</body>  
</html>

И снова, можно видеть пример обращения к вложенному слою:

document.layers["clippingLayer"].layers["imgLayer"].top= 100 + pos;

С остальными элементами этого скрипта Вы уже должны быть знакомы.

### Различные эффекты с вложенными слоями

Интересные эффекты могут быть созданы с помощью (частично) прозрачных слоев. Сочетание специально подобранных изображений с прозрачными областями может создавать совершенно потрясающий результат. Не все форматы изображений поддерживают работу с прозрачными частями. В настоящее время лучший из отвечающих этому условию форматов - gif89a. Большинство новых графических программ поддерживает этот формат. Помимо этого, в Internet доступны некоторые свободно распространяемые инструменты для работы с графикой.   
Новый формат изображений PNG также поддерживает эффект прозрачных частей изображения. Я полагаю, что в ближайшем будущем мы увидим множество страниц, использующих этот формат (точнее, как только большинство браузеров смогут его поддерживать). По сравнению с gif этот формат имеет множество преимуществ.

В данном примере используются два изображения (сплошные серые зоны здесь на самом деле являются прозрачными):

|  |  |
| --- | --- |
| http://shil.h1.ru/_private/JavaScript_main/JavaScript/imagesJs/davinci1.gif | http://shil.h1.ru/_private/JavaScript_main/JavaScript/imagesJs/davinci2.gif |

Сам скрипт несильно отличается от других примеров - так что я не буду здесь его распечатывать (впрочем, Вы конечно можете увидеть его, выбрав в меню Вашего браузера пункт *'View document source'*).

В Сети можно найти множество замечательных страниц, основанных на сочетании слоев с прозрачными частями. Некоторые из таких примеров Вы можете найти на моей**странице с примерами JavaScript** (она является частью home page моей книги о JavaScript и находится по адресу http://www.dpunkt.de/javascript /) - сама страница доступна как в [английском](http://www.dpunkt.de/javascript/engl/), так и в [немецком](http://www.dpunkt.de/javascript/) варианте.

Я надеюсь, что с помощью этого описания Вы получили представление об основных приемах использования слоев. Поэтому в будущем я надеюсь увидеть действительно прекрасные эффекты, созданные на основе JavaScript...

# Объект Image в JavaScript



Для **работы с изображениями в JavaScript** используется **объект Image**. Данный объект является очередным **свойством объекта Document**. И о том, как **управлять изображениями через JavaScript**, Вы узнаете из этой статьи.

**Конструктор у объекта Image** практически не используется (да и он классический). Методы у него присутсвуют (наследуются от **объекта Object**), но ничего интересного из себя не представляют. Поэтому в этой статье мы разберём самое важное в **объекте Image - его свойства**.

Прежде чем начать обрабатывать изображение необходимо его создать. Разумеется, создание происходит в **HTML**, поэтому знакомый Вам тег:

<img name = 'img' src = 'image1.jpg' />

Теперь мы можем обратиться к этому объекту через **JavaScript**:

document.write(document.img)

Как видите, **обращение к объекту Image** очень простое: сначала пишится **объект Document**, а затем его свойство с именем **объекта Image** (это имя мы задали в атрибуте "**name**"). В результате выполнения этого скрипта Вы увидите: "**[object HTMLImageElement]**". Это сработал **метод toString()**, но, впрочем, забудьте, что я сейчас написал - это тема будущих статей.

**ВНИМАНИЕ:** Необходимо соблюдать очень важное правило: **нельзя обращаться к тому, чего ещё не существует**. Какой вывод из этого можно сделать? Очень простой: пока не создано изображение, его нельзя обрабатывать. То есть Вы не можете запустить приведённый здесь скрипт **ДО** того, как появилось изображение. На это очень часто напарываются новички, поэтому не забывайте, что прежде чем работать с чем-либо, необходимо для начала это создать.

Теперь переходим к свойствам. Начнём со **свойства border**. Данное свойство отвечает за размер рамки вокруг изображения. Разумеется, мы его можем и прочитать, и записать. Давайте изменим размер рамки нашего изображения:

document.img.border = 5;

Разумеется, есть два свойства, отвечающие за ширину и высоту изображения. Это **свойства width и height объекта JavaScript**. Давайте их выведем:

document.write("Ширина изображения - " + document.img.width + "<br />");  
document.write("Высота изображения - " + document.img.height);

И последнее свойство, которое мы рассмотрим - это **src**. Данное свойство отвечает за путь к картинке. И давайте с Вами решим такую задачу: есть картинка и есть кнопка. При нажатии на кнопку картинка меняется.

<script language = 'javascript'>  
 var flag = 0;  
 function changeImage() {  
 if (flag == 0) {  
 document.img.src = 'image1.jpg';  
 flag = 1;  
 }  
 else {  
 document.img.src = 'image2.jpg';  
 flag = 0;  
 }  
 }  
</script>  
<img name = 'img' src = 'image1.jpg' />  
<form>  
 <input type = 'button' value = 'Сменить изображение' onClick = 'changeImage()' />  
</form>

Теперь поясню, как работает данный скрипт. Сначала мы описываем саму функцию. Создаётся переменная **flag**. Это некий флаг, который переключается при смене изображения. Дальше идёт функция **changeImage()**, которая и занимается сменой изображения. Изображению присваивается тот путь к картинке, которому соответствует флаг. После смены изображения меняется и значение флага (чтобы в следующий раз было другое изображение). За пределами скрипта создаётся форма с одной кнопкой. Здесь обратите внимание на атрибут "**onClick**". Этот атрибут отвечает за обработку события "*Клик мыши по кнопке*". О событиях мы поговорим отдельно, но здесь достаточно понять принцип. В значении атрибута "**onClick**" стоит функция, которую надо выполнить при нажатии на кнопку. Таким образом, у нас и меняется изображение. Надеюсь, что понятно объяснил.

Остальные **свойства объекта Image в JavaScript** используется достаточно редко, поэтому

**3.4 Библиотека JQuery: назначение и особенности использования**

**1. Введение в jQuery**

**jQuery** – библиотека JavaScript, которая позволяет получать доступ к любому элементу модели DOM, обращаться к атрибутам и содержимому, а также манипулировать ими. Кроме этого библиотека предоставляет удобный API для работы с технологией AJAX.

Библиотека была создана в 2006 году. В настоящее время практически каждый сайт использует данную библиотеку.

Подключив библиотеку jQuery вместо десятков команд на JavaScript можно написать несколько команд. Команды основаны на селекторах и классах CSS.

Библиотеку jQuery можно скачать с сайта [http://jquery.com/](http://aermolenko.ru/jexr/aHR0cDovL2pxdWVyeS5jb20v). Размер библиотеки в минимальном варианте составляет примерно 60 Кб.

Перед началом работы библиотеку надо подключить:

<script src=»jquery.js» type=»text/javascript»></script>

Вся работа с библиотекой ведется с использованием функции $. Общая идея использования: 1) выбирается элемент или группа элементов, 2) выполняются действия над выделенными элементами.

Пример. Выделение красным цветом каждой второй строки таблицы.

<script src=»jquery.js» type=»text/javascript»></script>

<style>

.rrr{color:red}

</style>

<body>

<table border=»1″>

<tr><td>1</td><td>1</td><td>1</td></tr>

<tr><td>1</td><td>1</td><td>1</td></tr>

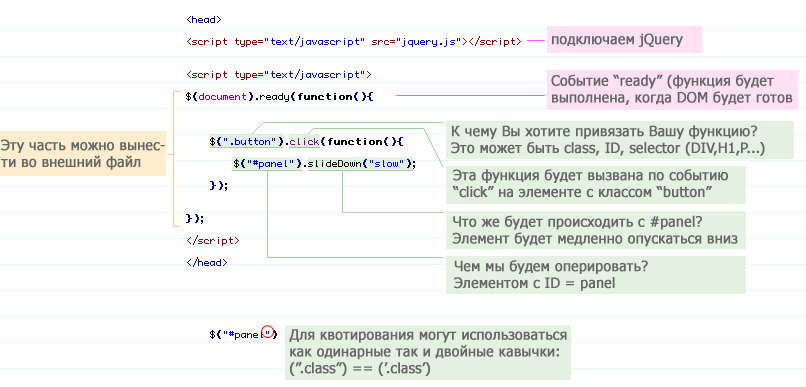
<tr><td>1</td><td>1</td><td>1</td></tr>

</table>

<script>

$(«table tr:even»).addClass(«rrr»)

</script>



Работу с jQuery можно разделить на 2 типа:

1. Получение jQuery-объекта с помощью функции $(). Например, передав в неё CSS-селектор, можно получить jQuery-объект всех элементов HTML, попадающих под критерий и далее работать с ними с помощью различных методов jQuery-объекта. В случае, если метод не должен возвращать какого-либо значения, он возвращает ссылку на jQuery объект, что позволяет вести цепочку вызовов методов согласно концепции текучего интерфейса.

2. Вызов глобальных методов у объекта $, например, удобных итераторов по массиву.

**2. Обращение к элементам**

Рассмотрим на примерах варианты обращения. Обращаю внимание, что все основано на синтаксисе CSS.

1. $(p a) выбирает все ссылки, расположенные в абзацах.

2. $(‘ #myId ‘) – выбор элемента с указанным идентификатором.

3. $(‘myClass’) – использование классов.

4. $(‘body>div’) – выбор элементов div, являющихся прямыми потомками элемента body.

5. $(p:even) – четные абзацы.

6. Использование ссылок.

* a[href^=http://] – этому селектору соответствуют все ссылки, которые начинаются с символов http://, на это указывает символ ^/
* a[href$=.pdf] – ссылки заканчиваются на «pdf»
* a[href\*=ya.ru] – ссылка содержит упоминание ya.ru в произвольном месте.

7. Имеется возможность выбора определенных элементов при условии, что они содержат другие элементы (в примере ниже двоеточие как в математике означает “такое что”). Например, селектор

li:has(a)

выбирает элементы <li>, которые содержат элемент <a>.

8. Выбор по позиции (некоторые варианты)

:first – первое совпадение на странице;

:last – последнее совпадение на странице;

:first-child – первый дочерний элемент;

:last-child – последний дочерний элемент;

:nth-child(n) – n-й элемент;

:nth-child(even|odd) – четные или нечетные дочерние элементы;

:even – четные элементы;

:odd – нечетные элементы.

9. Выбор элементов на основе характеристик, не предусмотренных спецификацией CSS

:input – выбирает элементы формы;

:selected – выбранные элементы option;

:visible – выбор невидимых элементов.

**3. Обработчик готовности документа**

Первый способ – запуск обработчика, когда загрузилась страница

window.onload=function(){

$(«table tr:nth-child(even) «).addClass(«even»);

};

Недостаток данного метода заключается в том, что ничего не будет изменено, пока все не загрузится, включая картинки, т.е. пользователь может усмотреть как все изменяется, т.е. увидеть и исходный вариант, и преобразованный.

Второй способ – дождаться только загрузки структуры документа, а именно:

$(documen).ready(function(){

$(«table tr:nth-child(even) «).addClass(«even»);

});

Т.е. читаем буквально – “Запуск функции после загрузки”. У этой конструкции имеется сокращенная форма:

$(function(){

$(«table tr:nth-child(even) «).addClass(«even»);

});

**4. Создание элементов DOM**

Создание элемента

$(‘<div>Hi</div>’)

Пустой элемент можно создать так:

$(‘<div> ‘)

Можно использовать кавычки, можно апострофы. Но наличие тегов обязательно. Задавать просто текст нельзя. Вспомните (лекция 7), сколько было команд, чтобы создать веточку элементов без jquery. Сама по себе эта команда ничего не выведет, надо эту веточку опять же привязать к родительскому элементу. Делается это так:

$(‘<div>Hi</div>’).insertAfter(‘#a1’)

Здесь a1 – это идентификатор объекта, после которого необходимо вставить элемент.

**5. Работа с полученным набором значений**

В терминах jQuery эти наборы называют обернутыми.

8.5.1. Определение размера. Для этого можно использовать свойство length, а можно метод size().

Пример. Замена содержимого с идентификатором a1 на количество элементов класса b2

$(«#a1»).html($(«.b2»).size())

8.5.2. Полученный набор значений рассматривается как массив, и с ним мы соответственно и работаем. Получить любой элемент в обернутом наборе можно по индексу. Например, получить второй элемент в наборе всех гиперссылок:

$(‘a’)[1].

Вместо индексов можно использовать метод get(), т.е. последнюю команду можно записать так:

$(‘a’).get(1).

При помощи метода get() можно получить обычный массив JavaScript, содержащий все обернутые элементы. Пример:

var allanchors=$(‘a’).get().

8.5.3. Существуют специальные команды библиотеки, которые позволяют объединять различные полученные наборы, убирать определенные значения по некоторым правилам

**6. Манипулирование объектами на странице**

Команды jQuery позволяют манипулировать свойствами, атрибутами, стилями и содержанием элементов.

Для обращения к свойствам и их значениям используются методы JavaScript, нет методов непосредственно библиотеки.

Пример. Изменяет свойство title у всех элементов класса b2

$(‘.b2’).each(function(n){this.title=»New «+n})

Здесь использована команда **each(функция).** Она выполняет обход всех элементов в наборе и вызывает для них функцию. В качестве параметра функции передается индекс элемента в наборе.

**7. Обработка событий**

Различные браузеры по-своему могут обрабатывать события, jQuery пытается сгладить эти неприятности. Поэтому обращаемся только к методам jQuery, а библиотека уже сама смотрит, что за браузер и применяет то или иное свойство.

* Модель событий jQuery обладает следующими свойствами:
* поддерживает единый метод установки событий;
* позволяет устанавливать несколько обработчиков для события;
* использует стандартные названия типов событий;
* предоставляет единые методы отмены события и блокирования действий по умолчанию.

Подключение обработчиков. Рассмотрим пример функции, которая будет срабатывать при щелчке по любому рисунку:

$(‘img’).blind(‘click’,fuction(event){alert(‘Приветствую!’);});

Для удаления обработчика используется команда unbind()

**8. Скрытие и отображение объектов**

8.1. Функции **hide**() и **show**(). С ними есть некоторые нюансы (например, show показывает изначальное состояние свойства display, поэтому при загрузке рекомендуют скрыть объект при помощи hide()), но в целом все как в примере.

Пример. $(‘#b3’).hide()

А затем, скажем, по щелчку, $(‘#b3’).show()

8.2. Имеются различные эффекты – раскрывающиеся списки, слайдеры, увеличение рисунков (когда-нибудь надо разобрать подробнее)

**9. Создания слайдера**

Рассмотрим пример создания слайдера – блока с контентом, выползающего по нажатию на конец закладки.

В текст вставляем следующий фрагмент:

<div id="container">  
<div id="panel">  
Содержимое панели  
</div>  
<p><a href="#">Справка</a></p>  
</div>

При помощи стилей изначально скрываем панель. Остальные параметры просто описательные – размер, цвет и т.д.

<!—Из-за наличия auto заодно и центрует в Mozila, т.к. auto Указывает, что размер отступов будет автоматически рассчитан браузером.—>

<!—buttontext можно вообще не задавать—>

<style>

.rrr {color:red}

#container {

margin: 0 auto;

width: 152px;

}

#panel {

background: #1ca8f6;

height: 230px;

display: none;

}

.button {

width: 152px;

height: 40px;

border-top: #333 dotted 1px;

text-align: center;

}

.buttontext {

font-weight: bold;

font-size: 1.2em;

text-shadow: 1px 1px 1px #666;

}

</style>

Описываем функцию

<script type="text/javascript">  
$(document).ready(function(){  
$(".buttontext").click(function(){  
$("#panel").slideToggle("normal"); return false;  
});  
});  
</script>

Комментарии

* Команда **.slideToggle** позволяет чередовать два простых действия: показать и скрыть. normal – это скорость, означает, что панель выедет за 400 миллисекунд, есть еще fast, slow.
* Команда **return false** означает, что мы не даем перейти по ссылке, в качестве которой используется кнопка-закладка. То есть, она просто работает в качестве, удобной ручки, за которую дергают, но не как непосредственно ссылка.