

Основы веб-технологий. ВолгГТУ. Кафедра ПОАС.

Составитель Овчинников С.А.

Лекция 1 Введение в веб-технологии

Лекция 2 Веб-дизайн и юзабилити.

Лекция 3 HTML и CSS.

Лекция 4 Язык PHP.

Лекция 5 СУБД Mysql

Лекция 6 Язык JavaScript. Возможности, особенности, ограничения. Эволюция и перспективы. Применение JavaScript для создания клиентской части веб-сайтов.

Вступление

При генерации страниц в Web возникает дилемма, связанная с архитектурой "клиент-сервер". Страницы можно генерировать как на стороне клиента, так и на стороне сервера. В **1995** году специалисты компании Netscape создали механизм управления страницами на клиентской стороне, разработав язык программирования JavaScript.

Таким образом, JavaScript - это язык управления сценариями просмотра гипертекстовых страниц Web на стороне клиента.

Основная идея JavaScript состоит в возможности изменения значений атрибутов HTML-контейнеров и свойств среды отображения в процессе просмотра HTML-страницы пользователем. При этом перезагрузки страницы не происходит.

На практике это выражается в том, что можно, например, изменить цвет фона страницы или интегрированную в документ картинку, открыть новое окно или выдать предупреждение.

Название "JavaScript" является собственностью Netscape. Реализация языка, осуществленная разработчиками Microsoft, официально называется Jscript. Версии JScript совместимы (если быть совсем точным, то не до конца) с соответствующими версиями JavaScript, т.е. JavaScript является подмножеством языка JScript.

JavaScript стандартизован ECMA (European Computer Manufacturers Association - Ассоциация европейских производителей компьютеров). Соответствующие стандарты носят названия ECMA-262 и ISO-16262. Этими стандартами определяется язык ECMAScript, который примерно эквивалентен JavaScript 1.1. Отметим, что не все реализации JavaScript на сегодня полностью соответствуют стандарту ECMA. В рамках данного курса мы во всех случаях будем использовать название JavaScript.

Понятие объектной модели применительно к JavaScript

Для создания механизма управления страницами на клиентской стороне было предложено использовать объектную модель документа. Суть модели в том, что каждый HTML-контейнер - это объект, который характеризуется тройкой:

- свойства
- методы
- события

Объектную модель можно представить как способ связи между страницами и браузером. Объектная модель - это представление объектов, методов, свойств и событий, которые присутствуют и происходят в программном обеспечении браузера, в виде, удобном для работы с ними кода HTML и исходного текста сценария на странице. Мы можем с ее помощью сообщать наши пожелания браузеру и далее - посетителю страницы. Браузер выполнит наши команды и соответственно изменит страницу на экране.

Объекты с одинаковым набором свойств, методов и событий объединяются в классы однотипных объектов. Классы - это описания возможных объектов. Сами объекты появляются только после загрузки документа браузером или как результат работы программы. Об этом нужно всегда помнить, чтобы не обратиться к объекту, которого нет.

Свойства

Многие HTML-контейнеры имеют атрибуты. Например, контейнер якоря `<A ...>...` имеет атрибут `HREF`, который превращает его в гипертекстовую ссылку:

```
<A HREF=intuit.htm>intuit</A>
```

Если рассматривать контейнер якоря `<A ...>...` как объект, то атрибут `HREF` будет задавать свойство объекта "якорь". Программист может изменить значение атрибута и, следовательно, свойство объекта:

```
document.links[0].href="intuit.htm";
```

Не у всех атрибутов можно изменять значения. Например, высота и ширина графической картинке определяются по первой загруженной в момент отображения страницы картинке. Все последующие картинки, которые заменяют начальную, масштабируются до нее. Справедливости ради следует заметить, что в Microsoft Internet Explorer размер картинке может меняться.

Для общности картины свойствами в JavaScript наделены объекты, которые не имеют аналогов в HTML-разметке. Например, среда исполнения, называемая объектом Navigator, или окно браузера, которое является вообще самым старшим объектом JavaScript.

Методы

В терминологии JavaScript методы объекта определяют функции изменения его свойств. Например, с объектом "документ" связаны методы open(), write(), close(). Эти методы позволяют сгенерировать или изменить содержание документа. Приведем простой пример:

```
function hello()
{ id=window.open("", "example", "width=400, height=150");
  id.focus(); id.document.open();
  id.document.write("<H1>Привет!</H1>");
  id.document.write("<HR><FORM>");
  id.document.write("<INPUT TYPE=button VALUE='Закреть окно' ");
  id.document.write("onClick='window.opener.focus();window.close(); '>");
  id.document.close();
}
```

В этом примере метод open() открывает поток записи в документ, метод write() осуществляет эту запись, метод close() закрывает поток записи в документ. Все происходит так же, как и при записи в обычный файл.

События

Кроме методов и свойств объекты характеризуются событиями. Собственно, суть программирования на JavaScript заключается в написании обработчиков этих событий. Например, с объектом типа button (контейнер INPUT типа button - "Кнопка") может происходить событие click, т.е. пользователь может нажать на кнопку. Для этого атрибуты контейнера INPUT расширены атрибутом обработки события click - onClick. В качестве значения этого атрибута указывается программа обработки события, которую должен написать на JavaScript автор HTML-документа:

```
<INPUT TYPE=button VALUE="Нажать" onClick="window.alert('Пожалуйста, нажмите еще раз');">
```

Обработчики событий указываются в тех контейнерах, с которыми эти события связаны. Например, контейнер BODY определяет свойства всего документа, поэтому обработчик события завершения загрузки всего документа указывается в этом контейнере как значение атрибута onLoad.

Примечание. Строго говоря, каждый браузер, будь то Internet Explorer, Netscape Navigator или Opera, имеет свою объектную модель. Объектные модели разных браузеров (и даже разные версии одного) отличаются друг от друга, но имеют принципиально одинаковую структуру. Поэтому нет смысла останавливаться на каждой из них по отдельности. Мы будем рассматривать общий подход применительно ко всем браузерам, иногда, конечно, заостряя внимание на различиях между ними.

Размещение кода на HTML-странице

Главный вопрос любого начинающего программиста: "Как оформить программу и выполнить ее?". Попробуем на него ответить как можно проще, но при этом не забывая обо всех способах применения JavaScript-кода.

Во-первых, исполняет JavaScript-код браузер. В него встроен интерпретатор JavaScript. Следовательно, выполнение программы зависит от того, когда и как этот интерпретатор получает управление. Это, в свою очередь, зависит от функционального применения кода. В общем случае можно выделить четыре способа функционального применения JavaScript:

- Лекция 7 гипертекстовая ссылка (схема URL);
- Лекция 8 обработчик события (handler);
- Лекция 9 подстановка (entity) (в Microsoft Internet Explorer реализована в версиях от 5.X и выше);
- Лекция 10 вставка (контейнер SCRIPT).

В учебниках по JavaScript описание применения JavaScript обычно начинают с контейнера SCRIPT. Но с точки зрения программирования это не совсем правильно, поскольку такой порядок не дает ответа на ключевой вопрос: как JavaScript-код получает управление? То есть каким образом вызывается и исполняется программа, написанная на JavaScript и размещенная в HTML-документе.

В зависимости от профессии автора HTML-страницы и уровня его знакомства с основами программирования возможны несколько вариантов начала освоения JavaScript. Если вы программист классического толка (C, Fortran, Pascal и т.п.), то проще всего начинать с программирования внутри тела документа, если вы привыкли программировать под Windows, то в этом случае начинайте с программирования обработчиков событий, если вы имеете только опыт HTML-разметки или давно не писали программ, то тогда лучше начать с программирования гипертекстовых переходов.

URL-схема JavaScript

Схема URL (Uniform Resource Locator) - это один из основных элементов Web-технологии. Каждый информационный ресурс в Web имеет свой уникальный URL. URL указывают в атрибуте HREF контейнера A, в атрибуте SRC контейнера IMG, в атрибуте ACTION контейнера FORM и т.п. Все URL подразделяются на схемы доступа, которые зависят от протокола доступа к ресурсу, например, для доступа к FTP-архиву применяется схема ftp, для доступа к Gopher-архиву - схема gopher, для отправки электронной почты - схема smtp. Тип схемы определяется по первому компоненту URL: `http://intuit.ru/directory/page.html`

В данном случае URL начинается с `http` - это и есть определение схемы доступа (схема `http`).

Основной задачей языка программирования гипертекстовой системы является программирование гипертекстовых переходов. Это означает, что при выборе той или иной гипертекстовой ссылки вызывается программа реализации гипертекстового перехода. В Web-технологии стандартной программой является программа загрузки страницы. JavaScript позволяет поменять стандартную программу на программу пользователя. Для того чтобы отличить стандартный переход по протоколу HTTP от перехода, программируемого на JavaScript, разработчики языка ввели новую схему URL - JavaScript:

```
<A HREF="JavaScript:JavaScript_код">...</A>
<IMG SRC="JavaScript:JavaScript_код">
```

В данном случае текст "JavaScript_код" обозначает программы-обработчики на JavaScript, которые вызываются при выборе гипертекстовой ссылки в первом случае и при загрузке картинки - во втором.

Например, при нажатии на гипертекстовую ссылку **Внимание!!!** можно получить окно предупреждения:

```
<A HREF="JavaScript:alert('Внимание!!!');"> Внимание!!!</A>
```

А при нажатии на кнопку типа submit в форме можно заполнить текстовое поле этой же формы:

```
<FORM NAME=f METHOD=post
  ACTION="JavaScript:window.document.f.i.VALUE='Нажали кнопку
Click';void(0);">
<INPUT TYPE=submit VALUE=Click>
</FORM>
```

В URL можно размещать сложные программы и вызовы функций. Следует только помнить, что схема JavaScript работает не во всех браузерах, а только в версиях Netscape Navigator и Internet Explorer, начиная с четвертой.

Обработчики событий

Такие программы, как обработчики событий (handler), указываются в атрибутах контейнеров, с которыми эти события связаны. Например, при нажатии на кнопку происходит событие click:

```
<FORM><INPUT TYPE=button VALUE="Кнопка"
onClick="window.alert('intuit');"></FORM>
```

Подстановки

Подстановка (entity) встречается на Web-страницах довольно редко. Тем не менее это достаточно мощный инструмент генерации HTML-страницы на стороне браузера. Подстановки используются в качестве значений атрибутов HTML-контейнеров. Например как значение по умолчанию поля формы, определяющего домашнюю страницу пользователя, будет указан URL текущей страницы:

```
<SCRIPT>
function l()
{
  str = window.location.href;
  return(str.length);
}
</SCRIPT>
<FORM><INPUT VALUE="{window.location.href}" SIZE="{l()}";">
</FORM>
<SCRIPT>
<!-- Это комментарий ...JavaScript-код...// -->
</SCRIPT>
<BODY>
... Тело документа ...
</BODY>
</HTML>
```

Вставка (контейнер SCRIPT - принудительный вызов интерпретатора)

Контейнер SCRIPT - это развитие подстановок до возможности генерации текста документа JavaScript-кодом. В этом смысле применение SCRIPT аналогично Server Side Includes, т.е. генерации страниц документов на стороне сервера. Однако здесь мы забежали чуть вперед. При разборе документа HTML-парсер передает управление интерпретатору после того, как встретит тег начала контейнера SCRIPT. Интерпретатор получает на исполнение весь фрагмент кода внутри контейнера SCRIPT и возвращает управление HTML-парсеру для обработки текста страницы после тега конца контейнера SCRIPT.

Контейнер SCRIPT выполняет две основные функции:

1. размещение кода внутри HTML-документа;

2. условная генерация HTML-разметки на стороне браузера.

Первая функция аналогична декларированию переменных и функций, которые потом можно будет использовать в качестве программ переходов, обработчиков событий и подстановок. Вторая - это подстановка результатов исполнения JavaScript-кода в момент загрузки или перезагрузки документа.

Размещение кода внутри HTML-документа

Собственно, особенного разнообразия здесь нет. Код можно разместить либо в заголовке документа, внутри контейнера HEAD, либо внутри BODY. Последний способ и его особенности будут рассмотрены в разделе "Условная генерация HTML-разметки на стороне браузера". Поэтому обратимся к заголовку документа.

Код в заголовке размещается внутри контейнера SCRIPT:

```
<HTML>
<HEAD>
<SCRIPT>
function time_scroll()
{
    d = new Date();
    window.status = d.getHours()+":"+d.getMinutes()+":"+d.getSeconds();
    setTimeout('time_scroll();',500);
}
</SCRIPT>
</HEAD>
<BODY onLoad=time_scroll()>
<CENTER>
<H1>Часы в строке статуса</H1>
<FORM>
<INPUT TYPE=button VALUE="Закрыть окно" onClick=window.close()>
</FORM>
</CENTER>
</BODY>
</HTML>
```

В этом примере мы декларировали функцию time_scroll() в заголовке документа, а потом вызвали ее как обработчик события load в теге начала контейнера BODY (onLoad=time_scroll()).

В качестве примера декларации переменной рассмотрим изменение статуса окна-потомка из окна-предка:

Создадим дочернее окно с помощью следующей функции, продекларировав ее, а затем и вызвав:

```
function sel()
{
    id = window.open("", "example", "width=500,height=200,status,menu");
```

```

    id.focus();
    id.document.open();
    id.document.write("<HTML><HEAD>");
    id.document.write("<BODY>");
    id.document.write("<CENTER>");
    id.document.write("<H1>Change text into child window.</H1>");
    id.document.write("<FORM NAME=f>");
    id.document.write("<INPUT TYPE=text NAME=t SIZE=20
MAXLENGTH=20 VALUE='This is the test'>");
    id.document.write("<INPUT TYPE=button VALUE='Close the window'
onClick=window.close()></FORM>");
    id.document.write("</CENTER>");
    id.document.write("</BODY></HTML>");
    id.document.close();
}
<INPUT TYPE=button VALUE="Изменить поле статуса в окне примера"
onClick="id.defaultStatus='Привет'; id.focus();">

```

Открывая окно-потомок, мы поместили в переменную `id` указатель на объект окно `id=window.open()`. Теперь мы можем использовать ее как идентификатор объекта класса `Window`. Использование `id.focus()` в нашем случае обязательно. При нажатии на кнопку "Изменить поле статуса в окне примера" происходит передача фокуса в родительское окно. Оно может иметь размер экрана. При этом изменения будут происходить в окне-потомке, которое будет скрыто родительским окном. Для того чтобы увидеть изменения, надо передать фокус. Переменная `id` должна быть определена за пределами каких-либо функций, что и сделано. В этом случае она становится свойством окна. Если мы поместим ее внутри функции открытия дочернего окна, то не сможем к ней обратиться из обработчика события `click`.

Иерархия классов

Объектно-ориентированный язык программирования предполагает наличие иерархии классов объектов. В JavaScript такая иерархия начинается с класса объектов `Window`, т.е. каждый объект приписан к тому или иному окну. Для обращения к любому объекту или его свойству указывают полное или частичное имя этого объекта или свойства объекта, начиная с имени объекта старшего в иерархии, в который входит данный объект:

Сразу оговоримся, что приведенная нами схема объектной модели верна для Netscape Navigator версии 4 и выше, а также для Microsoft Internet Explorer версии 4 и выше. Еще раз отметим, что объектные модели у Internet Explorer и Netscape Navigator совершенно разные, а приведенная схема составлена на основе их общей части.

Вообще говоря, JavaScript не является классическим объектным языком (его еще называют облегченным объектным языком). В нем нет наследования и полиморфизма. Программист может определить собственный класс объектов через оператор `function`, но чаще пользуется стандартными объектами, их конструкторами и вообще не применяет деструкторы классов. Это объясняется тем, что область действия JavaScript-программы обычно не распространяется за пределы текущего окна.

Иногда у разных объектов JavaScript бывают определены свойства с одинаковыми именами. В этом случае нужно четко указывать, свойство какого объекта программист хочет использовать. Например, Window и Document имеют свойство location. Только для Window это объект класса Location, а для Document - строковый литерал, который принимает в качестве значения URL загруженного документа.

Следует также учитывать, что для многих объектов существуют стандартные методы преобразования значений свойств объектов в обычные переменные. Например, для всех объектов по умолчанию определен метод преобразования в строку символов: toString(). В примере с location, если обратиться к window.location в строковом контексте, будет выполнено преобразование по умолчанию, и программист этого не заметит:

```
<SCRIPT>
document.write(window.location);
document.write("<BR>");
document.write(document.location);
</SCRIPT>
```

Однако разница все-таки есть, и довольно существенная. В том же примере получим длины строковых констант:

```
<SCRIPT>
w=toString(window.location);
d=toString(document.location);
h=window.location.href;
document.write(w.length);
document.write(d.length);
document.write(h.length);
</SCRIPT>
```

Результат исполнения получите сами.

Как легко убедиться, при обращении к свойству объекта типа URL, а свойство location как раз является объектом данного типа, длина строки символов после преобразования будет другой.

Класс объектов Window — это самый старший класс в иерархии объектов JavaScript. К нему относятся объект Window и объект Frame. Объект Window ассоциируется с окном программы-браузера, а объект Frame — с окнами внутри окна браузера, которые порождаются последним при использовании автором HTML-страниц контейнеров FRAMESET и FRAME.

Поле location

В поле `location` отображается URL загруженного документа. Если пользователь хочет вручную перейти к какой-либо странице (набрать ее URL), он делает это в поле `location`.

Вообще говоря, `Location` — это объект. Из-за изменений в версиях JavaScript класс `Location` входит как подкласс и в класс `Window`, и в класс `Document`. Мы будем рассматривать `Location` только как `window.location`. Кроме того, `Location` — это еще и подкласс класса `URL`, к которому относятся также объекты классов `Area` и `Link`. `Location` наследует все свойства `URL`, что позволяет получить доступ к любой части схемы `URL`.

Рассмотрим характеристики и способы использования объекта `Location`:

- свойства;
- методы;
- событий, характеризующих `Location`, нет.

Как мы видим, список характеристик объекта `Location` неполный.

Свойства

Предположим, что браузер отображает страницу, расположенную по адресу:

```
http://intuit.ru:80/r/dir/page?search#mark
```

Тогда свойства объекта `Location` примут следующие значения:

```
window.location.href = http://intuit.ru:80/r/dir/page?search#mark
window.location.protocol = http;
window.location.hostname = intuit.ru;
window.location.host = intuit.ru:80;
window.location.port = 80
window.location.pathname = /r/dir/;
window.location.search = search;
window.location.hash = mark;
```

Методы

Методы `Location` предназначены для управления загрузкой и перезагрузкой страницы. Это управление заключается в том, что можно либо перезагрузить документ (`reload`), либо загрузить (`replace`). При этом в историю просмотра страниц (`history`) информация не заносится:

```
window.location.reload(true);
window.location.replace('#top');
```

Метод `reload()` полностью моделирует поведение браузера при нажатии на кнопку `Reload` в панели инструментов. Если вызывать метод без аргумента или указать его равным `true`, то браузер проверит время последней модификации документа и

загрузит его либо из кеша (если документ не был модифицирован), либо с сервера. Такое поведение соответствует простому нажатию на кнопку Reload. Если в качестве аргумента указать false, то браузер перезагрузит текущий документ с сервера, несмотря ни на что.

Чтобы не возникло проблем с безопасностью браузера, путешествовать по History можно, только используя индекс URL. При этом URL, как текстовая строка, программисту недоступен. Чаще всего этот объект используют в примерах или страницах, на которые могут быть ссылки из нескольких разных страниц, предполагая, что можно вернуться к странице, из которой пример будет загружен:

```
<FORM><INPUT TYPE=button VALUE="Назад" onClick=history.back()></FORM>
```

Данный код отображает кнопку "Назад", нажав на которую мы вернемся на предыдущую страницу.

Управление окнами

Что можно сделать с окном? Открыть (создать), закрыть (удалить), положить его поверх всех других открытых окон (передать фокус). Кроме того, можно управлять свойствами окна и свойствами подчиненных ему объектов. Описанию основных свойств посвящен раздел "Программируем свойства окна браузера", поэтому сосредоточимся на простых и наиболее популярных методах управления окнами:

- alert();
- confirm();
- prompt();
- open();
- close();
- focus();
- setTimeout();
- clearTimeout().

window.alert()

Метод alert() позволяет выдать окно предупреждения:

```
<A HREF="javascript:window.alert('Внимание')">
Повторите запрос!</A>
```

Все очень просто, но нужно иметь в виду, что сообщения выводятся системным шрифтом, следовательно, для получения предупреждений на русском языке нужна локализованная версия ОС.

window.confirm()

Метод `confirm()` позволяет задать пользователю вопрос, на который тот может ответить либо положительно, либо отрицательно:

```
<FORM>
<INPUT TYPE=button VALUE="Вы знаете JavaScript?"
onClick="if(window.confirm('Знаю все')==true)
{ document.forms[0].elements[0].value='Да'; }
else {
  document.forms[0].elements[0].value='Нет';
};"><BR>
</FORM>
```

Все ограничения для сообщений на русском языке, которые были описаны для метода `alert()`, справедливы и для метода `confirm()`.

window.prompt()

Метод `prompt()` позволяет принять от пользователя короткую строку текста, которая набирается в поле ввода информационного окна:

```
<FORM>
<INPUT TYPE=button VALUE="Открыть окно ввода"
onClick="document.forms[0].elements[1].value=window.prompt('Введите
сообщение');">
<INPUT SIZE=30>
</FORM>
```

Введенную пользователем строку можно присвоить любой переменной и потом разбирать ее в JavaScript-программе.

window.open()

У этого метода окна атрибутов больше, чем у некоторых объектов. Метод `open()` предназначен для создания новых окон. В общем случае его синтаксис выглядит следующим образом:

```
open("URL", "window_name", "param,param,...", replace);
```

где: URL — страница, которая будет загружена в новое окно, `window_name` — имя окна, которое можно использовать в атрибуте TARGET в контейнерах A и FORM.

Таблица 2.

Параметры	Назначение
<code>replace</code>	Позволяет при открытии окна управлять записью в массив History
<code>param</code>	Список параметров
<code>width</code>	Ширина окна в пикселах
<code>height</code>	Высота окна в пикселах

toolbar	Создает окно с системными кнопками браузера
location	Создает окно с полем location
directories	Создает окно с меню предпочтений пользователя
status	Создает окно с полем статуса status
menubar	Создает окно с меню
scrollbars	Создает окно с полосами прокрутки
resizable	Создает окно, размер которого можно будет изменять

Приведем следующий пример:

```
<FORM>
<INPUT TYPE=button VALUE="Простое окно"
  onClick="window.open(
    'about:blank','test1',
    'directories=no,height=200,location=no,menubar=no,resizable=no,scrollbars=no,status=no,toolbar=no,width=200');
">
<INPUT TYPE=button VALUE="Сложное окно"
  onClick="window.open(
    'about:blank','test2',
    'directories=yes,height=200,location=yes,menubar=yes,resizable=yes,scrollbars=yes,status=yes,toolbar=yes,width=200');
">
</FORM>
```

Листинг 14.4.

При нажатии кнопки "простое окно" получаем окно со следующими параметрами:

- directories=no - окно без меню
- height=200 - высота 200 px
- location=no - поле location отсутствует
- menubar=no - без меню
- resizable=no - размер изменять нельзя
- scrollbars=no - полосы прокрутки отсутствуют
- status=no - статусная строка отсутствует
- toolbar=no - системные кнопки браузера отсутствуют
- width=200 - ширина 200

При нажатии кнопки "сложное окно" получаем окно, где:

- directories=yes - окно с меню
- height=200 - высота 200 px
- location=yes - поле location есть
- menubar=yes - меню есть
- resizable=yes - размер изменять можно
- scrollbars=yes - есть полосы прокрутки
- status=yes - статусная строка есть
- toolbar=yes - системные кнопки браузера есть
- width=200 - ширина 200

window.close()

Метод `close()` — это обратная сторона медали метода `open()`. Он позволяет закрыть окно. Чаще всего возникает вопрос, какое из окон, собственно, следует закрыть. Если необходимо закрыть текущее, то:

```
window.close();
self.close();
```

Если необходимо закрыть родительское окно, т.е. окно, из которого было открыто текущее, то:

```
window.opener.close();
```

Если необходимо закрыть произвольное окно, то тогда сначала нужно получить его идентификатор:

```
id=window.open();
...
id.close();
```

Как видно из последнего примера, закрывают окно не по имени (значение атрибута `TARGET` тут ни при чем), а используют указатель на объект.

window.setTimeout()

Метод `setTimeout()` используется для создания нового потока вычислений, исполнение которого откладывается на время (ms), указанное вторым аргументом:

```
idt = setTimeout("JavaScript_код", Time);
```

Типичное применение этой функции — организация автоматического изменения свойств объектов. Например, можно запустить часы в поле формы:

```
var flag=0;
var idp=null;
function myclock()
{
  if(flag==1)
  {
    d = new Date();
    window.document.c.f.value =
    d.getHours()+"."+d.getMinutes()+"."+d.getSeconds();
  }
  idp=setTimeout("myclock();", 500);
}
function flagss()
{
  if(flag==0) flag=1; else flag=0;
}
```

```

...
<FORM NAME=c>
Текущее время:<INPUT NAME=f size=8><INPUT TYPE=button VALUE="Start/Stop"
onClick="flagss ();myclock ();">
</FORM>

```

Листинг 14.6.

Нужно иметь в виду, что поток порождается всегда, даже в том случае, когда часы стоят. Если бы он создавался только при значении переменной flag равном единице, то при значении 0 он исчез бы, тогда при нажатии на кнопку часы продолжали бы стоять.

window.clearTimeout

Метод clearTimeout() позволяет уничтожить поток, вызванный методом setTimeout(). Очевидно, что его применение позволяет более эффективно распределять ресурсы вычислительной установки. Для того чтобы использовать этот метод в примере с часами, нам нужно модифицировать функции и форму:

```

var idp1 = null;
function start()
{
d = new Date();
window.document.c1.f1.value =
d.getHours()+":"+d.getMinutes()+":"+d.getSeconds();
idp1=setTimeout("start();",500);
}
function stop()
{
clearTimeout(idp1);idp1=null;
}
...
<FORM NAME=c1>
Текущее время:<INPUT NAME=f1 size=8>
<INPUT TYPE=button VALUE="Start" onClick="if(idp1==null)start();">
<INPUT TYPE=button VALUE="Stop" onClick="if(idp1!=null)stop();">
</FORM>

```

Листинг 14.7.

В данном примере для остановки часов используется метод clearTimeout(). При этом, чтобы не порождалось множество потоков, проверяется значение указателя на объект потока.

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=f_name METHOD=get
ACTION="javascript:void(0);">
<INPUT NAME=i_name SIZE=30 MAXLENGTH=30>
</FORM>

```

Поэтому в программах на JavaScript к ним обращаются по имени:

```
window.document.f_name.i_name.value="Текстовое поле";
```

Того же эффекта можно достичь, используя массив форм загруженного документа:

```
window.document.forms[0].elements[0].value="Текстовое поле";
```

В данном примере не только к форме, но и к полю формы мы обращаемся как к элементу массива.

Рассмотрим подробнее объект Form, который соответствует контейнеру FORM.

Свойства	Методы	События
<ul style="list-style-type: none">• action• method• target• elements[]• encoding	<ul style="list-style-type: none">• reset()• submit()	<ul style="list-style-type: none">• onReset• onSubmit

Сами по себе методы, свойства и события объекта Form используются нечасто. Их переопределение обычно связано с реакцией на изменения значений полей формы.

action

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

```
<FORM METHOD=post
  ACTION="javascript:window.alert('We use JavaScript-code as an URL');
        void(0);">
<INPUT TYPE=submit VALUE="Продемонстрировать JavaScript в action">
</FORM>
```

elements[]

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

```
<FORM NAME=fe>
<INPUT NAME=fe1 SIZE=30 MAXLENGTH=30>
<INPUT TYPE=button VALUE="Ввести текст по имени"
  onClick="document.fe.fe1.value='Ввести текст по имени';">
<INPUT TYPE=button VALUE="Ввести текст по индексу"
  onClick="document.fe.elements[0].value='Ввести текст по индексу';">
<INPUT TYPE=reset VALUE="Очистить">
</FORM>
```

Как видно из этого примера, индексирование полей в массиве начинается с цифры "0". Общее число полей в форме доступно как результат обращения

```
document.forms[i].elements.length.
```

reset()

Метод reset(), не путать с обработчиком события onReset, позволяет установить значения полей формы по умолчанию. При этом использовать кнопку типа Reset не требуется:

```
<FORM NAME=r>
```

```
<INPUT VALUE="Значение по умолчанию" SIZE=30 MAXLENGTH=30>
<INPUT TYPE=button VALUE="Изменим текст в поле ввода"
onClick="document.r.elements[0].value='Изменили текст';">
</FORM>
<A HREF="javascript:document.r.reset();void(0);">
Установили значение по умолчанию</A>
```

В данном примере по гипертекстовой ссылке происходит возврат в форме значения по умолчанию.

submit()

Метод submit() позволяет проинициировать передачу введенных в форму данных на сервер. При этом методом submit() инициируется тот же процесс, что и нажатием на кнопку типа Submit. Это позволяет отложить выполнение передачи данных на сервер:

```
<FORM NAME=s METHOD=post
ACTION="javascript:window.alert('Данные подтверждены');void(0);">
Введите цифру или букву:<INPUT SIZE=1 MAXLENGTH=1>
</FORM>
<A HREF="javascript:document.s.submit();">Отправить данные</A>
```

Вообще говоря, можно написать скрипт, который будет передавать данные без ведома пользователя, с помощью метода submit(). Однако браузер выдает предупреждение о таком поведении кода на странице.

onReset

Событие reset (восстановление значений по умолчанию в полях формы) возникает при нажатии на кнопку типа Reset или при выполнении метода reset(). В контейнере FORM можно переопределить функцию обработки данного события. Для этой цели в него введен атрибут onReset:

```
<FORM onReset="javascript:window.alert(
'Event Reset');return false;">
<INPUT VALUE="Значение по умолчанию">
<INPUT TYPE=reset VALUE="Восстановить">
</FORM>
```

В этом примере следует обратить внимание на то, что обработчик события reset возвращает логическое значение false. Это сделано для того, чтобы перехватить обработку события reset полностью. Если обработчик события возвращает значение false, то установка полей по умолчанию не производится; если обработчик событий возвращает значение true, то установка значений полей по умолчанию производится.

onSubmit

Событие submit возникает при нажатии на кнопку типа Submit, графическую кнопку (тип image) или при вызове метода submit(). Для переопределения метода

обработки события submit в контейнер FORM добавлен атрибут onSubmit. Функция, определенная в этом атрибуте, будет выполняться перед тем, как отправить данные на сервер. При этом в зависимости от того, что функция вернет в качестве значения, данные либо будут отправлены, либо нет.

```
function test()
{
if(parseInt(document.sub.digit.value).toString()=="NaN")
{
window.alert("Некорректные данные в поле формы.");
return false;
}
else
{
return true;
}
}
...
<FORM NAME=sub onSubmit="return test();" METHOD=post
ACTION="javascript:window.alert('Данные подтверждены');void(0);">
<INPUT NAME=digit SIZE=1 MAXLENGTH=1><INPUT TYPE=submit VALUE="Отправить">
</FORM>
```

Пример 15.3.

В этом примере следует обратить внимание на конструкцию return test();. Сама функция test() возвращает значения true или false. Соответственно, данные либо отправляются на сервер, либо нет.

Текст в полях ввода

Поля ввода (контейнер INPUT типа 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, вводя в эти поля промежуточные значения переменных и свойств объектов.

```
<FORM>Число гипертекстовых ссылок:
<INPUT SIZE=10 MAXLENGTH=10 VALUE="&{document.links.length};">
до момента обработки формы.
<INPUT TYPE=button VALUE="Число всех гипертекстовых ссылок в документе"
onClick="window.document.forms[0].elements[0].value=document.links.length;">
<INPUT TYPE=reset VALUE="Установить по умолчанию">
</FORM>
```

Пример 15.4.

В данном примере первое поле формы — это поле ввода. Используя подстановку, мы присваиваем ему значение по умолчанию, а потом при помощи кнопки изменяем это значение.

Объект Text (текстовое поле ввода) характеризуется следующими свойствами, методами и событиями:

Свойства	Методы	События
- defaultValue	- blur()	- onBlur
- form	- focus()	- onChange
- name	- select()	- onFocus
- type		
- value		

Свойства объекта Text — это стандартный набор свойств поля формы. В полях ввода можно изменять только значение свойства value.

Обычно при программировании полей ввода решают две типовых задачи: защита поля от ввода данных пользователем и реакция на изменение значения поля ввода.

Защита поля ввода

Для защиты поля от ввода в него символов применяют метод blur() в сочетании с обработчиком события onFocus:

```
<FORM>
<INPUT SIZE=10 VALUE="1-е значение"
      onFocus="document.forms[0].elements[0].blur();">
<INPUT TYPE=button VALUE=Change
      onClick="document.forms[0].elements[0].value=
'2-е значение';">
<INPUT TYPE=reset VALUE=Reset>
</FORM>
```

В этом примере значение поля ввода можно изменить, только нажав на кнопки Change и Reset. При попытке установить курсор в поле ввода он немедленно оттуда убирается, и таким образом, значение поля не может быть изменено пользователем.

Изменение значения поля ввода

Реакция на изменение значения поля ввода обрабатывается посредством программы, указанной в атрибуте onChange:

```
<FORM METHOD="post" onSubmit="return false;">
<INPUT SIZE="15" MAXLENGTH="15" VALUE="Тест"
      onChange="window.alert(document.forms[0].elements[0].value);">
<INPUT TYPE="button" VALUE="Изменить"
      onClick="document.forms[0].elements[0].value='Change';">
</FORM>
```

Если установить фокус на поле ввода и нажать Enter, ничего не произойдет. Если ввести что-либо в расположенное выше поле ввода, а потом нажать на Enter, то появится окно предупреждения с введенным текстом (для Netscape Navigator) или ничего не произойдет (для Internet Explorer последних версий). Если вы используете Internet Explorer последних версий, то окно предупреждения появится только после установки фокуса вне поля ввода. Это следует прокомментировать следующим образом: во-первых, обработчик onChange вызывается только тогда, когда ввод в поле закончен. Событие не вызывается при каждом нажатии на кнопки клавиатуры при вводе текста в поле. Во-вторых, обработчик события не вызывается при изменении значения атрибута VALUE из JavaScript-программы. В этом можно убедиться, нажав на кнопку Change - окно предупреждения не

открывается. Но если ввести что-то в поле, а после этого нажать на Change, окно появится.

Отметим, что он работает по-разному для Internet Explorer и Netscape Navigator, а именно по-разному обрабатывается событие onChange. Для Internet Explorer при любом изменении поля событие обрабатывается незамедлительно, для Netscape Navigator — после потери фокуса активным полем.

Кнопки

Использование кнопок в Web вообще немислимо без применения JavaScript. Создайте форму с кнопкой и посмотрите, что будет, если на эту кнопку нажать — кнопка продавливается, но ничего не происходит. Ни одно из стандартных событий формы (reset или submit) не вызывается. Конечно, данное замечание не касается кнопок Submit и Reset.

Кнопка вводится в форму главным образом для того, чтобы можно было обработать событие click:

```
<FORM>
<INPUT TYPE=button VALUE="Окно предупреждения"
      onClick="window.alert('Открыли окно');">
</FORM>
```

Текст, отображаемый на кнопке, определяется атрибутом VALUE контейнера INPUT. С этим атрибутом связано свойство value объекта Button. Любопытно, что, согласно спецификации, изменять значение данного атрибута нельзя. Однако в версии 4 Netscape Navigator и Internet Explorer это допустимо.

Следует отметить, что в Netscape Navigator размер кнопки фиксирован (первое значение должно быть самым длинным, иначе будет не очень красиво), а в Internet Explorer размер изменяется в зависимости от длины текста.

Картинки

Кнопки-картинки — это те же кнопки, но только с возможностью отправки данных на сервер. Собственно, такие кнопки в JavaScript составляют две разновидности контейнера INPUT: image и submit. В JavaScript объект, связанный с данными кнопками, называется Submit.

```
<FORM>
Активная кнопка:
<INPUT TYPE=image SRC=images.gif onClick="return false;">
</FORM>
```

Как мы уже отмечали, данный объект обладает теми же свойствами, методами и событиями, что и объект Button. Но вот реакция в разных браузерах при обработке событий может быть различной. Так, в событии onClick в Internet Explorer можно отменить передачу данных на сервер, выдав в качестве значения возврата false. Netscape Navigator на такое поведение обработчика события вообще не реагирует, и отменять передачу можно только в атрибуте onSubmit контейнера FORM:

```
<FORM onSubmit="return false">  
Активная кнопка:  
<INPUT TYPE=image SRC=images.gif border=0>  
</FORM>
```

Наиболее интересной особенностью графических кнопок является их способность передавать в запросе на сервер координаты точки, которую указал пользователь, нажимая на кнопку мышью. К сожалению, обработать такое поведение кнопки в JavaScript-программе не удастся.

Обмен данными

Передача данных на сервер из формы осуществляется по событию submit. Это событие происходит при одном из следующих действий пользователя:

- нажата кнопка Submit;
- нажата графическая кнопка;
- нажата клавиша Enter в форме из одного поля;
- вызван метод submit().

При описании отображения контейнера FORM на объекты JavaScript было подробно рассказано об обработке события submit. В данном разделе мы сосредоточимся на сочетании JavaScript-программ в атрибутах полей и обработчиках событий. Особое внимание нужно уделить возможности перехвата/генерации события submit.

Кнопка Submit

Кнопка Submit представляет собой разновидность поля ввода. Она ведет себя так же, как и обычная кнопка, но только еще генерирует событие submit (передачу данных на сервер). В этом, с точки зрения JavaScript-программирования, она абсолютно идентична графическим кнопкам:

```
<FORM>  
<INPUT TYPE=submit VALUE=submit>  
</FORM>
```

В данном примере мы просто перезагружаем страницу.

С точки зрения программирования наибольший интерес представляет возможность перехвата события submit и выполнение при этом действий, отличных от стандартных. Для этой цели у кнопки есть атрибут обработки события click (onClick):

```
<FORM>
<INPUT TYPE=submit VALUE=Submit onClick="return false;">
</FORM>
```

. При нажатии на кнопку перезагрузки страницы не происходит — передача данных на сервер отменена. Обработчик действует так же, как обработчик события submit в контейнере FORM.

Теперь можно написать собственную программу обработки события submit:

```
function my_submit()
{
if(window.confirm("Хотите перезагрузить страницу?")) return true;
else return false;
}
...
<FORM>
<INPUT TYPE=submit VALUE=Submit onClick="return my_submit();">
</FORM>
```

Если подтвердить необходимость перезагрузки страницы, она действительно будет перезагружена, а при отказе (cancel) вы вернетесь в текущую страницу без перезагрузки. Действия могут быть и более сложными. В любом случае, если функция обработки возвращает значение true, то передача данных на сервер (в нашем примере — перезагрузка страницы) происходит, иначе (значение false) — данные не передаются.

Единственное поле в форме

Если в форме присутствует одно-единственное поле, и мы в него осуществили ввод и после этого нажали Enter, то браузер сгенерирует событие submit:

```
<FORM onSubmit="window.alert('Сделано');return false;">
<INPUT SIZE=10 MAXLENGTH=10>
</FORM>
```

Перехватить такое событие и обработать можно только за счет программы обработки события submit в контейнере FORM, что и сделано в примере.

В этом примере, кроме поля ввода, в форме присутствует меню. Если менять значения выбранных альтернатив, то перезагрузки не происходит, но стоит изменить значение в поле ввода и нажать Enter, происходит submit, и система выдает окно предупреждения.

Метод submit()

Метод submit() — это метод формы. Если в программе вызывается метод submit, то данные из формы, к которой применяется данный метод, передаются на сервер. Усовершенствуем пример с полем ввода и меню выбора (прежде чем выбирать альтернативы, прочтите комментарий под примером):

```
<FORM onSubmit="window.alert('Сделано');return false;">
<INPUT SIZE=10 MAXLENGTH=10>
<SELECT onChange="form.submit();">
<OPTION>Вариант 1<OPTION>Вариант 2</SELECT>
</FORM>
```

При выборе альтернативы пользователь сразу инициирует обмен данными с сервером. Событие submit в данном случае обработчиком событий не перехватывается, в отличие от нажатия Enter. Такое поведение браузера довольно логично. Если программист вызвал метод submit(), то, наверное, он предварительно проверил данные, которые отправляет на сервер.

Cookies

Волшебные ключики, или cookies, не являются полями формы, но, тем не менее, отойдя от строгого рассмотрения иерархии объектов JavaScript, мы уделим им немного внимания, как одному из механизмов управления обменом данными. Основная функция cookie — поддержка сеанса работы между клиентом (браузером) и сервером.

cookie — это небольшой фрагмент текста, который передается от сервера браузеру и потом может быть возвращен обратно. Подробно о cookie рассказано в "Спецификации Cookie", которую можно найти в главе "Дополнения". Программа на JavaScript способна прочитать выставленное значение cookie и даже изменить его. Для этой цели используют свойство объекта DOCUMENT — cookie:

```
<FORM>
<INPUT TYPE=button VALUE="Показать Cookies"
onClick="window.alert(window.document.cookie);">
</FORM>
```

В данном случае cookies отображаются в виде одной большой строки со множеством значений. Свойство cookie документа можно переопределить:

```
function asign()
{
document.cookie="n1=3";
window.alert(document.cookie);
}
...
<FORM>
<INPUT TYPE=button VALUE="Изменить n1" onClick="asign()">
</FORM>
```

Как видно из примера, программисту не нужно выделять cookie из строки. Браузер рассматривает cookies как ассоциированный массив (хеш) и изменяет значение cookie по имени "ключика".

Наконец, cookie можно удалить. Если быть более точным — деактивировать, указав время его действия:

```
function change_cookies()
{
a = new Array();
c = new Date();
a = document.cookie.split(';');
document.cookie=a[0]+"; expires="+c.toGMTString()+"";
window.alert(document.cookie);
}
...
<FORM>
<INPUT TYPE=button VALUE="delete cookies" onClick="change_cookies()">
</FORM>
```

В данном случае мы "удаляем" cookie за счет параметра expires (времени, до которого cookie живет). Так как мы берем текущее время, то cookie исчезает из списка "ключиков". Многократно нажимая на кнопку, можно удалить все cookies для данной страницы.

Как мы уже знаем, функции в JavaScript используются для многократного выполнения одной и той же задачи. До сих пор функции всегда вызывались вручную с помощью скобок: myFunction(). Что, если потребуется вызвать функцию, когда пользователь выполняет определенную задачу? В JavaScript можно соединить функцию практически с любым событием, которое может породить пользователь. Давайте посмотрим это в действии и напишем функцию, которая подсчитывает, сколько раз пользователь щелкнул на странице.

```
<script type="text/javascript">
var clickCount = 0;
function documentClick(){
  document.getElementById('clicked').value = ++clickCount;
}
document.onclick = documentClick;
</script>
```

Вы щелкнули на этой странице <input id="clicked" size="3" onfocus="this.blur();" value="0"> раз.

Вы щелкнули на этой странице раз.

В [предыдущей лекции](#) оператор ++ был применен только после переменной, как в случае "clickCount++". Однако в данном примере оператор ++ используется перед переменной. В первом примере "clickCount++", единица добавляется к переменной clickCount после чтения ее значения. В случае "++clickCount" единица добавляется к переменной clickCount перед чтением ее значения. Так как в этом примере

переменной `clickCount` в начале присваивается значение 0, то единицу к ней необходимо добавлять до задания значения поля ввода, поэтому использована запись `++clickCount`.

Преыдуший пример может показаться достаточно знакомым. Так же, как и раньше, определяется переменная и функция. Изменение состоит в том, что вместо вызова функции `documentClick()` код содержит указание, что функция должна выполняться всякий раз, когда пользователь щелкает на документе. `"document.onclick"` связывает функцию с событием документа `onclick` ("при щелчке").

Существует множество событий подобных `"onclick"`. Мы познакомимся с некоторыми из них, но наиболее распространенными являются: `onclick`, `onload`, `onblur`, `onfocus`, `onchange`, `onmouseover`, `onmouseout` и `onmousemove`. Функцию можно связать с событиями любого объекта, такого, как изображение или поле ввода, а не только документа. Например, события `onmouseover` и `onmouseout` используются обычно с изображениями для создания эффекта изменения.

Можно также заметить, что ссылка на поле ввода делается другим образом. Ранее говорилось, что для указания поля необходимо использовать `"document.forms.имяФормы.elements.имяПоляВвода"`. Хотя этот способ прекрасно работает, это не всегда необходимо. В данном примере поле ввода действует просто как счетчик. Оно не находится внутри формы, и нам это и не нужно. Поэтому мы задаем для поля некоторый ID (идентификатор): `id="clicked"`. ID можно использовать для ссылки на любой объект на странице. ID должен быть уникальным на странице, поэтому если имеется 5 полей ввода с ID, то все ID должны быть различны, даже если они только имеют вид `"input1"-->"input5"`.

Поскольку это поле ввода используется как счетчик, то нежелательно, чтобы пользователи щелкали на нем и изменяли его значение. Здесь на помощь приходит другое связывание, `"onfocus"`, которое срабатывает, когда курсор перемещается на объект. Поэтому при щелчке на поле ввода или при перемещении на поле ввода с помощью клавиши `Tab` вызывается `"onfocus"`.

Событие `onfocus` имеет очень короткий код, но он также очень важен. В нем появляется ключевое слово `"this"`, которое важно понимать в JavaScript. Ключевое слово `"this"` указывает на тот объект, на котором выполняется код. В данном примере `"this"` указывает на поле ввода. Выражение `"this.blur()"` "размывает" поле ввода, другими словами, заставляет его терять фокус ввода. Так как это происходит, как только пользователь активизирует поле ввода, то это делает "невозможным" изменение данных.

Если указатель `"this"` используется в функции, то он указывает на саму функцию. Если `"this"` используется в коде JavaScript вне функции, то он указывает на объект окна. Наиболее часто `"this"` используется для изменения свойства текущего объекта, как в примере выше, или для передачи текущего объекта функции.

Давайте посмотрим на другой пример:

```
<script type="text/javascript">
function showValue(obj){
    alert('You Clicked On ' + obj.value);
}
</script>

<input type="radio" name="fruit" onclick="showValue(this);" value="Яблоко" >
Яблоко
<input type="radio" name="fruit" onclick="showValue(this);" value="Апельсин"
> Апельсин
<input type="radio" name="fruit" onclick="showValue(this);" value="Груша" >
Груша
<input type="radio" name="fruit" onclick="showValue(this);" value="Банан">
Банан

Яблоко  Апельсин  Груша  Банан
```

Можно видеть, что событие `onclick` для каждой из этих радио-кнопок одинаково. Однако, если щелкнуть на каждой из них, то будут получены разные сообщения. Это связано с использованием `"this"`. Так как `"this"` указывает на каждую отдельную радио-кнопку, то каждая радио-кнопка передается в функцию `"showValue"` по отдельности.

Вернемся к функциям и рассмотрим передачу функции аргументов. В предыдущем примере `"obj"` является аргументом. Предполагается, что `"obj"` содержит указатель на поле ввода, на котором был произведен щелчок. В функцию можно передавать любое количество аргументов. Если потребуется 10 аргументов, то функция будет выглядеть следующим образом:

```
function myFunction(arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8, arg9,
arg10){
    // здесь располагается код
}
```

Во многих случаях может понадобиться функция, которой требуется определенное количество аргументов, но не всегда требуются все. В JavaScript не нужно передавать все 10 аргументов в функцию, которая объявлена с 10 аргументами. Если передать только первые 3, то функция будет иметь только 3 определенных аргумента. Это необходимо учитывать при написании функций. Например, можно написать функцию, которой всегда требуются 3 первых аргумента, но следующие 7 являются необязательными:

```
function myFunction(arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8, arg9,
arg10){
    // код с arg1
    // код с arg2
    // код с arg3
    if(arg4){
        // код с arg4
    }

    if(arg5 && arg6 && arg7){
```

```

    // код с arg5, arg6 и arg7
    if(arg8){
        // код с arg8
    }
}

if(arg9 || arg10){
    // код с arg9 или arg10
}
}

```

Можно видеть, что в коде выполняется простая проверка "if(arg) ". Если аргумент не передается в функцию, то при попытке использовать этот аргумент будет получено значение "undefined ". При использовании "undefined" в качестве логического (булевого) значения (true / false), как в операторах if, оно воспринимается как false.

Поэтому, если arg4 не был передан в приведенном выше примере, то он является "undefined" и тело оператора if не выполняется.

Как быть, когда трудно определить, сколько потребуется аргументов? Можно иметь функцию, которая получает от 1 до n аргументов и выполняет с каждым из них одну и ту же задачу. На этот случай JavaScript имеет в каждой функции объект "arguments". Объект arguments содержит все аргументы функции:

```

function myFunction(){
    for(var i=0; i<arguments.length; i++){
        alert(arguments[i].value);
    }
}

```

Этот код сообщит значение всех переданных ему объектов. Если передать 100 объектов, то будет получено 100 сообщений. Более полезной функцией было бы, возможно, скрывание/вывод всех переданных функции объектов.

Один из наиболее интересных аспектов JavaScript - идея о том, что функции являются объектами и могут передаваться как поле ввода, изображение или что-то еще, что может быть. Посмотрите, например, следующий код:

```

<script type="text/javascript">
function multiply(){
    var out=1;
    for(var i=0; i<arguments.length; i++){
        out *= arguments[i];
    }

    return out;
}

function add(){
    var out=0;
    for(var i=0; i<arguments.length; i++){
        out += arguments[i];
    }
}

```

```

    }

    return out;
}

function doAction(action){
    alert(action(1, 2, 3, 4, 5));
}
</script>

<button onclick="doAction(multiply) ">Test Multiply</button>
<button onclick="doAction(add) "      >Test Add</button>

Test Multiply          Test Add

```

В этом небольшом фрагменте кода происходит очень многое. Вначале просто определяют две функции: `multiply` и `add`. Функция `multiply` просто перемножает все переданные ей числа. Аналогично, функция `add` складывает все переданные ей числа. Тонкости начинаются при использовании действий `onclick` двух созданных кнопок. Можно видеть, что при щелчке на любой из двух кнопок в функцию `doAction` передается объект. Ранее мы всегда передавали переменные или объекты HTML (такие, как поля ввода в предыдущем примере). В этом примере передаются функции. Функции можно передавать таким же способом, как и любой другой объект, и когда они передаются, их можно вызывать таким же способом, как и любую другую функцию. Изменяется только ее имя.

Таким образом функция `doAction` получает другую функцию в качестве аргумента! Если передается функция `multiply`, то функция `doAction` передает ей значения от 1 до 5, и мы получаем в результате 120. Если в `doAction` передается функция `add`, то ей также передаются значения от 1 до 5, и в результате мы получаем значение 15.

Это в действительности одно из наиболее мощных свойств JavaScript, и оно будет более подробно рассмотрено в следующих лекциях. Пока достаточно понять общий принцип.

Другим важным свойством функций является возможность вложения их друг в друга. JavaScript не поддерживает истинный объектно-ориентированный подход к проектированию, но это свойство обеспечивает очень похожие возможности.

Кроме вложения функций, важно отметить, что имеется несколько различных способов объявления функций:

```

function myFunction(){
    function nestedFunction1(arg1, arg2, arg3){
        alert(arg1+arg2+arg3);
    }

    var nestedFunction2 = function(arg1, arg2, arg3){
        alert(arg1+arg2+arg3);
    }
}

```

```
var nestedFunction3 = new Function('arg1, arg2, arg3',  
'alert(arg1+arg2+arg3);');  
}
```

В этом примере функции `nestedFunction1`, `nestedFunction2` и `nestedFunction3` являются одинаковыми по своим возможностям. Единственное различие состоит в том, как они определяются. `nestedFunction1` объявлена, как это делалось раньше. Синтаксис для `nestedFunction2` немного отличается. Мы задаем для функции переменную `this.nestedFunction2`. Синтаксис этого объявления будет следующий "имяПеременной=function(аргументы){". Аналогично для функции `nestedFunction3` задается переменная для новой функции. Однако это объявление существенно отличается, так как мы определяем функцию с помощью строк. Третий вариант используется редко, но является очень полезным, когда используется. Он позволяет создать строку, содержащую код выполняемой функции, а затем определить функцию с помощью этой строки.

Строки

В JavaScript строка является любым фрагментом текста. Как и многие другие объекты в JavaScript, строки можно определять несколькими различными способами:

```
var myString = 'Hello, World!';  
var myString = new String('Hello, World!');
```

Первый метод используется наиболее часто. Вторым методом применяется редко и только для гарантии, что получаемый объект является строкой. Например:

```
var n = 5;  
var s = new String(n*20);
```

В этом примере `s` будет строкой "100". Если просто задать `s` как `n*20`, то `s` будет содержать число 100. Однако поскольку JavaScript является слабо типизированным языком, то эти различия не будут существенно влиять на то, что вы делаете.

Строковые объекты (`var n = new String('Hello World')`) технически являются в Internet Explorer более медленными при некоторых операциях, чем строковые литералы (`var n = 'Hello World'`). Однако это поведение совершенно противоположно в других браузерах. В любом браузере различие редко бывает настолько заметно, чтобы об этом беспокоиться.

Единственное важное различие состоит в том, что `eval()` не работает со строковыми объектами.

Что, если в строке имеется апостроф? Следующий код работать не будет:

```
var n = 'The dog took it's bone outside';
```

Легко видеть, что апостроф в "it's " заканчивает строку. Поэтому мы получаем строку "The dog took it ", за которой следует " s bone outside' ". Это продолжение само по себе не является допустимым кодом JavaScript (или правильным грамматически, если на то пошло), поэтому будет получена ошибка.

Здесь можно сделать две вещи. Так как строка определяется с помощью одиночных или двойных кавычек, то можно задать строку с помощью двойных кавычек. Другая возможность состоит в экранировании апострофа. Чтобы экранировать символ, необходимо просто подставить перед ним символ \. Символ \ в этом контексте сообщает JavaScript, что следующий символ необходимо воспринимать в точности так, как он есть, в номинальном значении, а не как специальный символ.

```
var n = "The dog took it's bone outside";  
var n = 'The dog took it\'s bone outside';
```

Если в строке должен присутствовать символ \, то он экранируется таким же образом: '\\'.
var n = '\\';

В [предыдущей лекции](#) мы встречались с функциями indexOf и lastIndexOf. Напомним, что они делают. Функция indexOf возвращает число, определяющее первую позицию одной строки в другой. Если разыскиваемая строка не существует, то indexOf возвращает -1. Функция lastIndexOf идентична indexOf, но возвращает не первую позицию вхождения строки, а последнюю.

Тот факт, что функции indexOf и lastIndexOf возвращают -1, если строка не существует, является очень полезным и позволяет использовать эти функции для достаточно распространенной задачи - проверки того, что одна строка существует внутри другой.

Существует несколько других полезных функций для работы со строками, которые мы перечислим и кратко поясним.

- charAt() сообщает, какой символ находится в определенной позиции строки. Поэтому 'Test'.charAt(1) = 'e'.
- length сообщает длину строки . 'Test'.length = 4.
- substring() выдает строку между двумя индексами. 'Test'.substring(1, 2) = 'e'.
- substr() аналогична substring(), только второе число является не индексом, а длиной возвращаемой строки. Если это число указывает на позицию за пределами строки, то substr() вернет существующую часть строки. 'Test'.substr(1, 2) = 'es';
- toLowerCase() и toUpperCase() делают то, что обозначают: преобразуют строку в нижний или верхний регистр символов соответственно. 'Test'.toUpperCase() = 'TEST';

Примеры всех приведенных выше функций:

```
alert('This is a Test'.indexOf('T')); // 0
alert('This is a Test'.lastIndexOf('T')); // 10
alert('This is a Test'.charAt(5)); // i
alert('This is a Test'.length); // 14
alert('This is a Test'.substring(5, 9)); // is a
alert('This is a Test'.substr(5, 9)); // is a Test
alert('This is a Test'.toUpperCase()); // THIS IS A TEST
alert('This is a Test'.toLowerCase()); // this is a test
```

Последней строковой функцией, которая будет рассмотрена, является `eval()`. `eval()` получает строку и выполняет строку, как если бы это был код JavaScript.

```
eval("alert('Hello, World!')");
```

В этом примере будет выведено сообщение "Hello, World!", как если бы функция `alert` была написана обычным образом. Это может быть очень полезно, так как позволяет создать содержащую код строку, а затем ее выполнить.

Числа

Работа с числами в JavaScript происходит достаточно просто. В лекциях [1](#) и [2](#) было показано, как выполняются базовые арифметические операции, операторы `++`, `--`, а также `*=`, `+=`, `/=` и `-=`. Мы узнали, что `NaN` означает "Не число" и что делает функция `isNaN()`. Осталось рассмотреть еще только несколько вещей.

Объект `Math` в JavaScript содержит функции, позволяющие сделать почти все, что можно сделать с числами помимо обычной арифметики. `Math.PI`, например, содержит просто число 3.141592653589793. В нем содержатся тригонометрические функции (`sin`, `cos`, `tan`, и т.д.), функции для округления чисел (`Math.floor` возвращает число, округленное с недостатком, `Math.ceil` возвращает число, округленное с избытком, а `Math.round` округляет число "нормально") и многие другие. Существует очень много функций, объяснять которые здесь не имеет смысла. Их всегда можно найти в подходящем справочнике. (Например, <http://www.devguru.com/technologies/javascript/10734.asp>)

Двумя наиболее распространенными задачами, связанными с числами, являются преобразование числа в строку и строки в число. Как уже говорилось, JavaScript является слабо типизированным языком, а это означает, что типы данных не имеют большого значения, но существуют некоторые случаи, когда надо быть уверенным, что имеется число или строка. Если надо сложить, например, 5 и число, которое вводит пользователь, то надо быть уверенным, что введено число, а не слово "Привет".

```
var n = parseInt("3.14"); // 3
var n = parseFloat("3.14") // 3.14
```

Функция `parseInt` возвращает целое значение своего аргумента. Аргументы "3.14 ", "3 ", "3.00001 " и "3.9999 " превратятся в значение 3. Функция `parseFloat`, с другой стороны, возвращает также любое десятичное значение. Обе эти функции пытаются "очистить" данные перед возвращением числа. Например, `parseInt("3a")` вернет значение 3.

Существует также несколько методов, которые можно использовать, когда надо преобразовать число в строку:

```
var n = 5;

var m = n.toString();
var m = n+'';
var m = new String(n);
```

Как говорилось ранее, последний метод может быть немного непривычным, поэтому предполагается, что пользователь будет держаться от него в стороне, если только не понадобится использовать объект `String` для специальных целей. Предпочтительным методом является "`n.toString()`", но необходимо отметить, что часто используется второй метод. Например, если имеется уведомление `alert("Имеется ' + apples + ' яблок')`, то число `apples` автоматически преобразуется в строку.

Если необходимо выполнить строковую операцию с переменной, то необходимо быть уверенным, что имеется строка. Если, например, имеется запись года из 4 цифр и ее надо преобразовать в 2 цифры:

```
var year = 2000;

var sYear = year.toString();
var year2 = year.substr(year.length-2);
```

Можно было бы также легко вычесть 2000 из этой даты, но что, если датой является 1995? или 1800? или 2700 или просто 5? В результате могли бы получиться совершенно неправильные даты, если вычесть 2000 из каждой такой даты. Используемый метод всегда даст правильные две цифры года.

Массивы

Массив является по сути списком элементов. Каждый элемент массива может быть чем угодно, но обычно они связаны друг с другом. Если, например, необходимо отследить 30 студентов класса, то можно создать массив студентов:

```
var students = new Array();
students[0] = 'Sam';
students[1] = 'Joe';
students[2] = 'Sue';
students[3] = 'Beth';
```

Как можно видеть, определить массив очень просто, так же, как и присвоить значения его элементам. Однако пример выше имеет слишком много кода для относительно небольшого результата. Нет ничего удивительного в том, что существует несколько методов для создания массива. Значительно более компактным будет следующий пример:

```
var students = ['Sam', 'Joe', 'Sue', 'Beth'];
```

Это код создает точно такой же массив, что и предыдущий пример, но, как можно видеть, он значительно более компактный и ничуть не сложнее для понимания. Скобки ([и]) в этом примере сообщают коду, что будет создан массив. Простая запись "var students = [];" является тем же самым, что и запись "var students = new Array();". Правда, некоторые люди считают, что использование слова "Array" более наглядно, чем запись "[]", поэтому можно использовать любой метод записи.

Что можно теперь сделать с этим массивом студентов? Чтобы обратиться к любому элементу массива, необходимо знать его индекс. В первом примере можно видеть, что в скобках находятся числа (0-3). Это индексы. Если требуется узнать имя третьего студента, то надо будет написать "alert(students[2]);". Почему 2, а не 3? Массивы в JavaScript начинают индексацию с 0, а не с 1. Поэтому первым элементом в нашем массиве будет students[0], вторым - students[1], сотым - students[99], и т.д. Для этого не существует никакой реальной причины, просто так устроен JavaScript и многие другие языки программирования. Некоторые другие языки используют в качестве первого индекса 1.

Наиболее распространенной задачей при работе с массивами, помимо изменения его данных, является проверка его длины, обычно для того, чтобы можно было перебрать весь массив и выполнить некоторую задачу с каждым элементом.

```
var students = ['Sam', 'Joe', 'Sue', 'Beth'];
var suffixes = ['1st', '2nd', '3rd', '4th'];

for(var i=0; i<students.length; i++){
    alert(suffixes[i]+' студент -- '+students[i]);
}
```

Важный момент, который необходимо знать о массивах, состоит в том, что каждый элемент массива может содержать любой произвольный объект. В этих примерах каждый элемент массива является строкой, но они могут быть также числами, объектами, функциями, даже другими массивами. Электронная таблица (такая, как Excel) является хорошим примером массива, содержащего другие массивы. Прежде всего имеется массив столбцов. Каждый столбец будет в свою очередь содержать в себе массив строк. Этот массив создается точно таким же образом, как и массив students:

```
var spreadsheet = [
    ['A1', 'B1', 'C1', 'D1'],
    ['A2', 'B2', 'C2', 'D2'],
    ['A3', 'B3', 'C3', 'D3'],
    ['A4', 'B4', 'C4', 'D4']
]
```

```
];
```

Переносы строк в JavaScript обычно не имеют значения. В этом примере переносы строк используются для придания коду большей наглядности и не влияют на код никаким образом.

Можно видеть, что здесь имеется 5 массивов. Четыре внутренних массива (или вложенных массива) содержатся в одном большом массиве, `spreadsheet`. Если потребуется узнать значение на пересечении второго столбца и третьей строки, то можно написать:

```
var col2 = spreadsheet[1];
alert(col2[2]);

// или

alert(spreadsheet[1][2]);
```

Оба фрагмента кода делают одно и то же, выводят значение "C2".

Существует несколько распространенных операций, которые выполняются с массивами. Первой является добавление элемента в конце массива. Вернемся к массиву `students`, который содержит в данный момент 4 элемента. Чтобы добавить новый элемент, надо просто задать значение для 5-го элемента:

```
var students = ['Sam', 'Joe', 'Sue', 'Beth'];

students[4] = 'Mike';
students[students.length] = 'Sarah';
students.push('Steve');

// теперь массив содержит 7 элементов: ['Sam', 'Joe', 'Sue', 'Beth', 'Mike',
'Sarah', 'Steve']
```

Здесь также существует несколько способов для выполнения этой задачи. Первый метод, `students[4]`, используется редко, так как обычно неизвестно заранее в точности, сколько будет элементов. Поэтому применяется один из двух оставшихся методов. `push` является функцией, которая просто добавляет то, что получает, в конец массива, как и предыдущий метод, использующий свойство `.length`.

Не так часто, но иногда необходимо также удалить объект из массива. В этом случае задействуется функция `splice`, которая позволяет добавить или удалить любое количество элементов массива, но в данный момент мы собираемся использовать ее для удаления одного студента, `Mike`, который переехал в другой город:

```
var students = ['Sam', 'Joe', 'Sue', 'Beth', 'Mike', 'Sarah', 'Steve'];
students.splice(4, 1);
```

Splice в этом примере получает два аргумента: начальный индекс и число элементов для удаления. Так как Mike является пятым студентом, то его индекс будет 4. Будет удален только 1 студент, поэтому здесь используется 1. В результате имеем массив с удаленным Mike:

```
['Sam', 'Joe', 'Sue', 'Beth', 'Sarah', 'Steve'];
```

Чаще всего точно неизвестно, где в массиве находится элемент. К сожалению, единственным способом выяснить это является перебор всех элементов массива. Можно написать небольшой простой сценарий, который позволит легко добавлять или удалять студентов:

```
var students = ['Sam', 'Joe', 'Sue', 'Beth'];

function addStudent(name) {
    students.push(name);
}

function removeStudent(name) {
    for(var i=0; i<students.length; i++){
        if(students[i].toLowerCase() == toLowerCase(name)){
            students.splice(i, 1);
            break;
        }
    }
}
```

Имя студента:

Добавить этого студента
Удалить этого студента

Студенты:

Единственным новым моментом здесь является слово "break". break останавливает выполнение кода любого цикла, в котором находится: цикла for, цикла do или switch. Поэтому в данном случае, когда удаляемый студент найден, мы прерываем цикл for, так как выполнили свою задачу.

Часто бывает необходимо преобразовать массив в строку или строку в массив. Имеется две функции, которые могут легко это сделать: join и split. Функция join получает массив и преобразует его в строку с помощью разделителя, заданного в join. Функция split действует в обратном направлении и делает массив из строки, определяя новый элемент с помощью разделителя, заданного в split:

```
var myString = 'apples are good for your health';
var myArray = myString.split('a');
// строка myString разбивается на элементы на каждом найденном
символе 'a'.
alert(myArray.join(', '));
// преобразуем myArray снова в строку с помощью запятой,
// так что можно видеть каждый элемент
alert(myArray.join('a'));
```

```
'a',  
    // теперь преобразуем myArray снова в строку с помощью символа  
    // так что снова получается исходная строка
```

Еще две полезные функции для работы с массивами - "pop" и "shift". Функция "pop" удаляет последний элемент из массива и возвращает его. Функция "shift" удаляет первый элемент из массива и возвращает его.

```
var students = ['Sam', 'Joe', 'Sue', 'Beth'];  
  
while(students.length>0){  
    alert(students.pop());  
}
```

К сожалению, при этом массив был уничтожен: он теперь пуст. Иногда именно это и надо сделать. Если требуется просто очистить массив, то проще всего задать его длину (length) равной 0:

```
students.length = 0
```

Теперь массив пуст. Даже если снова задать длину массива больше 0, все данные в массиве уже будут уничтожены.

Все использованные до сих пор массивы называются "индексными массивами", так как каждый элемент массива имеет индекс, который необходимо использовать для доступа к элементу. Существуют также "ассоциативные массивы", в которых каждый элемент массива ассоциирован с именем в противоположность индексу:

```
var grades = [];  
grades['Sam'] = 90;  
grades['Joe'] = 85;  
grades['Sue'] = 94;  
grades['Beth'] = 82;
```

Ассоциативные массивы действуют немного иначе, чем индексные. Прежде всего, длина массива в этом примере будет равна 0. Как же узнать, какие элементы находятся в массиве? Единственный способ сделать это - использовать цикл "for-in":

```
for(student in grades){  
    alert("Оценка " + student + "будет: " + grades[student]);  
}
```

Синтаксис цикла "for-in" следующий: "for(item in object){". Цикл пройдет через все элементы в объекте, и элемент будет именем элемента. В данном случае элементом является "Sam", затем "Joe", "Sue" и "Beth".

Последнее замечание о массивах состоит в том, что в действительности можно объединять ассоциативные и индексные массивы, хотя это обычно не

рекомендуется, так как может вызывать некоторые проблемы. При правильном использовании, однако, можно с успехом это применять.

```
var students = ['Sam', 'Joe', 'Sue', 'Beth'];

students['Sam'] = 90;
students['Joe'] = 85;
students['Sue'] = 94;
students['Beth'] = 82;

alert('Всего имеется '+(students.length)+' студентов: '+students.join(', '));
for(var i=0; i<students.length; i++){
    alert("Оценка " +students[i]+"будет: "+students[students[i]]);
}
```

Хотя это может показаться немного сложным, здесь нет ничего такого, о чем не говорилось в этой лекции.

Прежде всего, необходимо понять, что с точки зрения браузера страница HTML является точно тем же, что и документ XML. Если читатель имеет опыт работы с XML, то сможет понять обработку DOM достаточно легко. Но в любом случае это в действительности не сложно. Существует прекрасный справочник по адресу (http://www.devguru.com/technologies/xml_dom/index.asp), который подробно описывает каждый метод обработки DOM, но к концу этой лекции читатель в основном поймет, как это работает.

Необходимо также отметить атрибуты в некоторых из этих тегов. Например, тег TABLE (<table border="0" cellspacing="2" cellpadding="5">) имеет 3 заданных атрибута: border, cellspacing и cellpadding. При изменении DOM часто бывает необходимо изменить эти атрибуты. Можно, например, изменить атрибут SRC тега IMG, чтобы изменить выводимое изображение. Это часто делают, например, для создания эффекта изменения изображения, на которое направлен указатель (rollover).

Создание эффекта изменения изображения

Теперь, имея общее представление о компоновке страницы, можно начинать ее модификацию. Начнем с создания простого эффекта изменения изображения:

```

```

В этом коде присутствуют 4 события изображения: onmouseover, onmousedown, onmouseout и onmouseup. Каждое из этих событий имеет присоединенный простой фрагмент кода JavaScript, который изменяет атрибут src изображения. Создавая три разных изображения, можно легко и быстро создать изображение с тремя сменяющимися друг друга состояниями.

Добавление и удаление элементов

Одной из задач, которая становится все более распространенной в современных приложениях JavaScript, является возможность добавления или удаления элементов страницы. Предположим, что имеется форма, которая позволяет послать кому-нибудь ссылку. Обычно используется одно поле ввода для адреса e-mail и второе - для имени получателя. Если требуется послать ссылку нескольким адресатам, то либо придется посылать форму несколько раз, либо можно было бы разместить на странице более одного набора полей имя/e-mail. Но в этом случае мы все еще ограничены заданным числом контактов. Если имеется пространство для 5 контактов и необходимо послать ссылку 20 людям, то форму придется заполнять 4 раза.

JavaScript позволяет обойти эту проблему, делая возможным динамическое дополнение и удаление содержимого страницы:

```
1  var inputs = 0;
2  function addContact() {
3      var table = document.getElementById('contacts');
4
5      var tr    = document.createElement('TR');
6      var td1   = document.createElement('TD');
7      var td2   = document.createElement('TD');
8      var td3   = document.createElement('TD');
9      var inp1  = document.createElement('INPUT');
10     var inp2  = document.createElement('INPUT');
11
12     if(inputs>0) {
13         var img    = document.createElement('IMG');
14         img.setAttribute('src', 'delete.gif');
15         img.onclick = function() {
16             removeContact(tr);
17         }
18         td1.appendChild(img);
19     }
20
21     inp1.setAttribute("Name", "Name" +inputs);
22     inp2.setAttribute("Name", "Email"+inputs);
23
24     table.appendChild(tr);
25     tr.appendChild(td1);
26     tr.appendChild(td2);
27     tr.appendChild(td3);
28     td2.appendChild(inp1);
29     td3.appendChild(inp2);
30
31     inputs++;
32 }
33 function removeContact(tr) {
34     tr.parentNode.removeChild(tr);
35 }
36 <table>
37     <tbody id="contacts">
38         <tr>
39             <td colspan="3"><a href="javascript:addContact();">Добавьте
контакт</a></td>
40         </tr>
41         <tr>
42             <td></td>
```

```
43         <td>Name </td>
44         <td>Email</td>
45     </tr>
46 </tbody>
47 </table>
```

Возможно вам не приходилось ранее использовать тег TBODY. Многие браузеры автоматически добавляют этот тег в DOM, не сообщая об этом. Если необходимо изменить содержимое таблицы, то в действительности необходимо изменить содержимое TBODY. Во избежание возможных недоразумений мы просто добавили тег TBODY, чтобы каждый мог бы его видеть. Все это может показаться достаточно сложным, но здесь очень мало нового материала.

Прежде всего здесь имеется новая функция: `document.createElement`. Функция `createElement` создает задаваемый аргументом элемент. Можно видеть, что в строках сценария с 5 по 10 создается несколько элементов. В действительности создается новая строка TR, которая вставляется в таблицу в строках 37-46. В результате новая строка TR будет выглядеть следующим образом:

```
<tr>
  <td>
    
  </td>
  <td>
    <input name="Name1">
  </td>
  <td>Если это
    <input name="Email1">
  </td>
</tr>
```

Другими словами, мы создали 7 элементов: 1 TR, 3 TD, 2 INPUT и 1 IMG. Тег IMG будет использоваться как изображение кнопки "Удалить". Так как пользователь должен всегда видеть по крайней мере 1 строку ввода, то первую строку удалить невозможно. Поэтому в 12 строке сценария проверяется, что создается первая строка. Если строка не первая, то добавляется изображение.

После создания всех этих элементов остается в действительности поместить их в документ. Каждый элемент на странице имеет встроенную функцию " `appendChild` ", которую можно использовать для добавления к этому элементу потомка. Когда добавляется потомок, то он добавляется как последний элемент, поэтому если таблица уже имеет в качестве потомков 10 тегов TR и добавляется еще один, то он будет добавлен как 11-ый тег TR. Мы начинаем с получения ссылки на таблицу (строка 3). Затем мы добавляем TR к этой таблице (строка 24) и добавляем затем 3 TD (строки 25-27). Второй и третий TD содержат поле ввода, поэтому мы добавляем эти поля ввода (28-29).

Вот и все! Теперь у нас есть новый элемент TR, и он находится на странице. Осталось пояснить еще пару моментов. Чтобы форма была обработана правильно, все поля ввода должны иметь различные имена. Поэтому мы задаем имя двух полей ввода на основе счетчика (21-22), а затем увеличиваем счетчик (31). Это делается с помощью еще одной новой функции `setAttribute`, которая

имеет два параметра: имя атрибута и значение атрибута. Для нее существует дополнительная функция `getAttribute`, которая имеет только один аргумент: имя атрибута, значение которого надо получить.

```
element.setAttribute("name", "elementName")
```

по сути то же самое, что

```
element.name="elementName"
```

Однако задание атрибута непосредственно, как в предыдущем примере, может иногда вызывать некоторые проблемы для различных браузеров или для некоторых специфических атрибутов. Поэтому хотя любой метод обычно будет работать, предпочтительным является первый метод, использующий `setAttribute`.

Необходимо также позаботиться о кнопке удаления. Мы уже знаем, что кнопка удаления для первой строки полей не создается, но необходимо заставить ее работать для всех остальных. Это делается в строках кода 15-16. Здесь к изображению добавлена функция `onclick`, которая вызывает функцию `removeContact`, передавая элемент `TR` в качестве единственного аргумента.

Взглянув на функцию `removeContact`, можно видеть, что сначала происходит обращение `"tr.parentNode"` к функции `parentNode`, которая является еще одной функцией для работы с DOM. Она просто возвращает порождающий элемент для текущего элемента. Если посмотреть на изображенное ранее дерево документа, то видно, что `parentNode` вернет элемент непосредственно над элементом, на котором он вызван. Поэтому, если вызвать `parentNode` на одиночном элементе `A` в этом дереве, то будет получена ссылка на элемент `TD` над ним.

Поэтому `tr.parentNode` возвращает ссылку на элемент `TABLE` над `TR`. Затем вызывается функция `removeChild` на этом элементе `TABLE`, которая просто удаляет у предка указанного потомка.

Взглянув еще раз на строку 34, можно теперь увидеть, что она просто говорит: "Удалить элемент `TR` у его предка" или еще проще "Удалить элемент `TR`".

Элементы потомки

Ко всем потомкам элемента можно обратиться с помощью атрибута `childNodes`, который возвращает массив, содержащий все узлы потомки текущего элемента. Можно также использовать атрибуты `firstChild` и `lastChild` на любом элементе, чтобы получить ссылки на первый или на последний элемент.

Чтобы увидеть, как это работает, давайте напишем сценарий для раскраски чередующихся строк `TR` в таблице:

```

function setColors(tbody, color1, color2){
    var colors = [color1, color2];
    var counter = 0;
    var tr      = tbody.firstChild;

    while(tr){
        tr.style.backgroundColor = colors[counter++ % 2];
        tr = tr.nextSibling;
    }
}

```

При рассмотрении этого небольшого фрагмента кода мало что нужно пояснять в том, как можно получить этот интересный небольшой эффект. Код начинается с получения ссылки на первый элемент TR в таблице с помощью метода `firstChild`. Затем каждый TR раскрашивается по очереди двумя разными цветами, используя `tr.style`. Цвет фона задается одним из двух цветов из массива `colors`. Если `counter` имеет четное значение, то цвет фона задается как `color1`. Иначе он задается как `color2`. Это реализуется с помощью оператора деления по модулю (%). Для тех, кто забыл, напомним, что операция вычисляет остаток при делении. $5/2 = 2$ с остатком 1. Поэтому $5 \% 2$ (5 по модулю 2) = 1.

Здесь не будет обсуждаться в данный момент изменение стилей, но достаточно сказать, что `element.style` предоставляет доступ ко всему, что можно задать с помощью таблицы стилей. Если нужно, например, задать стиль элемента, то можно прочитать/записать весь стиль с помощью `element.style.cssText`.

После задания цвета фона берется следующий элемент TR в таблице. Это делается с помощью функции `nextSibling`, которая возвращает следующий элемент в DOM, с тем же предком, что и текущий элемент. Если посмотреть на `table`, то все его потомки являются элементами TR, поэтому `nextSibling` будет в цикле перебирать все элементы TR. Если отыскивается элемент TR с потомками, состоящими из элементов TD, то `nextSibling` будет циклически перебирать все элементы TD. Когда элементов TR больше не останется, цикл автоматически закончится, так как TR будет неопределенным, что в JavaScript оценивается как `false`.

С целью рассмотрения оператора `childNodes` и функции `getElementsByTagName` перепишем приведенный пример немного по-другому:

```

function setColors(tbody, color1, color2){
    var colors = [color1, color2];

    for(var i=0; i<tbody.childNodes.length; i++){
        tbody.childNodes[i].style.backgroundColor = colors[i % 2];
    }
}

function setColors(tbody, color1, color2){
    var colors = [color1, color2];
    var trs    = tbody.getElementsByTagName('TR');
}

```

```
for(var i=0; i<trs.length; i++){
  trs[i].style.backgroundColor = colors[i % 2];
}
}
```

Обе эти функции делают то же самое, что и первая функция `setColors`, но написано это немного по-другому. Первая функция использует атрибут `childNodes`. Как ранее говорилось, `childNodes` содержит массив, элементами которого являются потомки. Поэтому мы циклически перебираем `tbody.childNodes` и изменяем цвет каждого потомка, которые все должны быть элементами `TR`.

Другая функция использует новую функцию `getElementsByTagName`, которая выдает массив всех элементов с указанным именем тега. Так как нам требуются все элементы `TR`, то мы просто передаем в функцию `'TR'` и получаем список всех элементов `TR` в таблице. После этого код почти идентичен предыдущей функции.

Работа с текстом

Работа с текстом немного отличается от работы с другими элементами `DOM`. Первое: каждый фрагмент текста на странице помещен в невидимый узел `#TEXT`. Поэтому следующий код HTML

```
<div id="ourTest">this is <a href="link.html">a link</a> and an image: </div>
```

имеет четыре корневых элемента: текстовый узел со значением `"this is "`, элемент `A`, еще один текстовый узел со значением `" and an image: "` и, наконец, элемент `IMG`. Элемент `A` имеет конечный текстовый узел в качестве потомка со значением `" a link "`. Когда необходимо изменить текст, то прежде всего необходимо получить этот "невидимый" узел. Если мы хотим изменить текст `" and an image: "`, то необходимо написать:

```
document.getElementById('ourTest').childNodes[2].nodeValue = 'our new text';
```

`document.getElementById('ourTest')` дает нам тег `div`. `childNodes[2]` дает узел текста `" and an image: "` и наконец `nodeValue` изменяет значение этого узла текста.

Что, если требуется добавить к этому еще текст, но не в конце, а перед `" a link "`?

```
var newText = document.createTextNode('our new text');
var ourDiv = document.getElementById('ourTest');
ourDiv.insertBefore(newText, ourDiv.childNodes[1]);
```

Первая строка показывает, как создать текст с помощью `document.createTextNode`. Это аналогично функции использованной ранее функции `document.createElement`. Третья строка содержит еще одну новую функцию `insertBefore`, которая аналогична `appendChild`, за исключением того, что имеет два аргумента: добавляемый элемент и существующий элемент, перед которым надо сделать вставку. Так как мы хотим добавить новый текст перед элементом `A` и знаем, что

элемент А является вторым элементом в div, то мы используем `ourDiv.childNodes[1]` в качестве второго аргумента для `insertBefore`.

По большей части это все манипуляции с DOM. Если требуется создать, например, поле с изменяемым размером, то для изменения ширины и высоты поля будут использоваться те же функции мыши и функции `getAttribute` и `setAttribute`. Очень похожим образом, если изменять верхнюю и левую позицию стиля элемента, то можно перемещать элементы по странице, либо в ответ на ввод мыши (перетаскивание), либо по таймеру (анимация).

В качестве последнего замечания к этой лекции: одним из наиболее полезных средств при попытке протестировать или отладить код JavaScript, который изменяет DOM, является сценарий обхода дерева DOM. Проще говоря - это сценарий, который показывает каждый элемент и каждый атрибут объекта DOM. Описание этого кода выходит за рамки этой лекции, но он мог бы, например, показывать все атрибуты и объекты-потомки любого получаемого в качестве аргумента объекта.

Как упоминалось в [шестой лекции](#), каждый объект на странице является потомком или предком какого-то другого объекта. Все это представляет большое дерево. Объект `window` находится в вершине этого дерева, а все остальное содержится в нем. Объект `window` указывает на реальное окно браузера. Если требуется, например, открыть новое окно или изменить размер текущего, то для этого используются функции объекта `window`.

Объект `window`, кроме того, что находится в вершине DOM, является также глобальным объектом. Во [второй лекции](#) мы говорили о том, что любая переменная обладает глобальной или локальной областью действия. Если она имеет глобальную область действия, то она доступна в любом месте сценария JavaScript. Обладание глобальной областью действия означает также, что переменная соединена непосредственно с объектом `window`. Любой код JavaScript, который не находится внутри какой-то функции, находится в глобальном объекте `window`.

В связи с этим не требуется писать `"window"` при обращении к функциям или переменным объекта `window`, как в случае использования некоторых других функций, таких, как `document.getElementById.alert()` является примером функции, которую можно вызвать либо как `"window.alert()"`, либо просто `"alert()"`.

Объект Window

Прежде всего объект `window` предоставляет доступ к информации об окне:

Свойство	Описание
<code>window.location</code>	возвращает текущий URL окна
<code>window.opener</code>	Если окно было открыто другим окном (с помощью <code>window.open</code>),

	то возвращается ссылка на открывающее окно, иначе null
MSIE: window.screenTop	Возвращает верхнюю позицию окна. Отметим, что эти значения в MSIE существенно отличаются от других браузеров. MSIE
Другие: window.screenY	возвращает верхнюю позицию области содержимого (ниже адресной строки, кнопок, и т.д.). Другие браузеры возвращают верхнюю позицию реального окна (выше кнопки закрытия)
MSIE: window.screenLeft	Возвращает левую позицию окна с такими же различиями, как и для screenTop и screenY
Другие: window.screenX	

Положение окна на экране пользователя редко бывает необходимо, но два других свойства, location и opener, будут очень полезны. Свойство window.location выполняет две функции. Если изменить его с помощью JavaScript, например, window.location='http://www.htmlgoodies.com', то браузер будет перенаправлен на эту страницу. Чтение window.location выдает адрес текущего документа. Зачем это нужно знать? Обычно адрес страницы не нужен, но может потребоваться строка запроса или анкер. Возьмем, например, следующий URL:

<http://www.somesite.com/page.asp?action=browse&id=5#someAnchor>. Этот URL можно разбить на три части:

URL:	http://www.somesite.com/page.asp
Строка запроса:	action=browse&id=5
Анкер:	someAnchor

Так как `window.location` содержит всю эту информацию, то можно написать функцию, которая будет возвращать переменную `queryString` (строку запроса). Это аналогично `request.querystring` в ASP или `$_GET` в PHP, если вы знакомы с каким-либо из этих языков:

```
function queryString(val){
    var q = unescape(location.search.substr(1)).split('&');

    for(var i=0; i<q.length; i++){
        var t=q[i].split('=');
        if(t[0].toLowerCase()==val.toLowerCase()) return t[1];
    }
    return '';
}
```

Для предыдущего URL функция `queryString('action')` вернет `'browse'`. Мы видим здесь новую функцию `"window.unescape"`. Функция `unescape`, а также ее дополнительная функция `escape`, используются в соединении с `window.location`. При передаче данных в строке запроса она должна быть экранирована ("escaped"), чтобы она не влияла на саму строку запроса. Если, например, среди данных имеется знак амперсанд (&), то необходимо его экранировать, чтобы можно было различить этот знак в данных и тот &, который разделяет два различных значения. Функция `escape` подготавливает посылаемые данные для использования в качестве значения `queryString`, поэтому она используется при задании `window.location`. Например:

```
window.location='/page.asp?name='+escape(SomeInputBox.value);
```

Функция `unescape` делает обратное и позволяет получить "нормальный" текст из `window.location`.

Вернемся к свойствам `window`, где имеется свойство `"opener"`. Это свойство используется в соединении с обычно используемой функцией `window.open`, которая позволяет открывать новое окно браузера и, для некоторых свойств управлять его выводом. Блокировщики всплывающих окон очень часто будут препятствовать открытию окна с помощью `window.open`, если в этот процесс не вовлечен щелчок мышью. Поэтому, если в коде имеется вызов `window.open` и при этом пользователь не щелкает на ссылке или чем-то подобном, то скорее всего это не будет работать.

Функция `window.open` получает до 3 аргументов: URL окна, которое надо открыть, имя окна и свойства окна.

```
var newWindow=window.open('', 'TestWindow', 'width=200,height=200');
newWindow.document.write('Это окно будет закрыто через 2 секунды');
setTimeout(function(){ newWindow.close(); }, 2000);
```

Третий аргумент `window.open` получает строку аргументов. Обычно используются следующие:

Лекция 11 width, height - задают размеры окна;
Лекция 12 left, top - задают положение окна на экране;
Лекция 13 location, menubar, toolbar, status, titlebar, scrollbars - эти параметры выводят/скрывают свои соответствующие "панели" на окне. Задайте "yes", чтобы вывести соответствующую "панель";
Лекция 14 resizable - если задан как "yes", то размер окна можно изменять.

Полное описание window.open можно увидеть в документации Mozilla.

Так как мы открываем пустое окно, то первый аргумент будет пустым. Для открытия страницы 'test.html' вызов выглядел бы следующим образом:
window.open ('test.html', 'TestWindow', 'width=200,height=200').

В этом примере для объекта window, открываемого окна, задается переменная newWindow. В связи с этим для вывода содержимого в окне необходимо использовать "newWindow.document.write".

Функция window.open также имеет свою противоположность, функцию window.close. Однако эта функция может успешно вызываться только на окнах, созданных JavaScript. Если попробовать закрыть окно, созданное не JavaScript, то возможны два варианта: либо появится сообщение, говорящее, что сценарий пытается закрыть окно, либо браузер просто это проигнорирует.

setTimeout и setInterval

Можно видеть, что в этом примере используется еще одна новая функция setTimeout. Функции setTimeout и setInterval применяются для выполнения кода после указанного интервала времени и обе получают два аргумента: функцию или строку кода и период ожидания в мс. 1 мс = 1/1000 секунды, поэтому для задания выполнения кода через 5 секунд необходимо определить в этом случае для второго аргумента значение 5000.

setTimeout выполнит код один раз после завершения заданного интервала времени. setInterval будет продолжать выполнять код после завершения каждого интервала. При заданном интервале 5000 setInterval будет выполнять код каждые 5 секунд.

Существуют еще две другие функции: clearTimeout и clearInterval, которые отменяют выполнение, через заданные интервалы. Однако для этого необходимо иметь ссылку на вызов setTimeout или setInterval, например:

```
var myTimeout = setTimeout("alert('Hi!');", 500);  
clearTimeout(myTimeout);
```

Если не сохранить ссылку в переменной myTimeout, то не существует способа отменить заданное выполнение. Давайте посмотрим на пример этого в действии:

```
function createTimeout(text, time){
  setTimeout("alert('"+text+"');", time);
}

var intervals = [];
function createInterval(text, time){
  // сохраняет интервал в массиве intervals
  intervals.push(setInterval("alert('"+text+"');", time));
}

function tut5(){
  if(intervals.length==0) return;
  // удаляет последний интервал выполнения в массиве intervals
  clearInterval(intervals.pop());
}
```

Демонстрация в действии

Текст для вывода:

Время ожидания(в мс):

```
setTimeout
setInterval
clearInterval
```

Существует также функция `clearTimeout`, которая идентична по синтаксису `clearInterval`.

Важно отметить, что во время ожидания выполнения заданного кода функциями `setTimeout` или `setInterval` весь остальной код JavaScript продолжает выполняться. Когда функция `setTimeout` или `setInterval` будет готова, она выполнит заданный код, но только после того, как другой код закончит выполнение. Другими словами, `setTimeout` и `setInterval` никогда не прерывают для выполнения другой код.

window.opener

Как говорилось ранее, свойство окна `'opener'` можно использовать для доступа к окну, которое открыло текущее окно, а также к любым свойствам, функциям и т.д. этого окна. Например:

```
<!--page1.html-->
<HTML>
<HEAD>
<script type="text/javascript">
window.open('page2.html', 'TestWindow',
'width=500,height=200,resizable=yes');
</script>
</HEAD>
</HTML>
```

```
<!--page2.html-->
<HTML>
<HEAD>
<script type="text/javascript">
document.write('URL окна предка будет : '+window.opener.location);
</script>
</HEAD>
</HTML>
```

Отметим, что это работает, только если URL открываемого окна находится на том же сервере, что и текущая страница. Если необходимо открыть, например, окно на <http://www.webreference.com>, то мы не получим доступ к свойствам этого окна. Это поддерживается всеми основными браузерами по соображениям безопасности.

Объект Document (window.document)

Одной из наиболее часто используемых функций в JavaScript является `document.write`. Можно сказать, что `document.write` получает строку и выводит ее на странице. Здесь необходимо только следить за одной вещью. Если страница полностью загрузилась и вызывается `document.write`, то вся страница будет очищена и на экране будет только результат работы `document.write`.

Мы уже видели различные свойства объекта `document` в действии. Например, `document.forms` возвращает массив всех форм на странице. Здесь также существует несколько свойств, подобных этому.

3. `document.forms` - массив, содержащий все формы на текущей странице;
4. `document.images` - массив, содержащий все изображения на текущей странице;
5. `document.links` - массив, содержащий все ссылки на текущей странице;
6. `document.anchors` - массив, содержащий все анкеры на текущей странице;
7. `document.applets` - массив, содержащий все апплеты на текущей странице;
8. `document.styleSheets` - массив, содержащий все таблицы стилей на текущей странице;
9. `window.frames` - массив, содержащий все фреймы на текущей странице.

Как мы видели в [предыдущей лекции](#), почти все эти свойства можно продублировать с помощью `document.getElementsByTagName`. Чтобы получить все изображения на странице, можно воспользоваться, например, `document.getElementsByTagName('IMG')`. Существует три подобные функции:

- `document.getElementById` - возвращает один элемент на основе его ID;
- `document.getElementsByName` - возвращает массив элементов, определенных по имени. В отличие от ID многие элементы могут иметь на странице одинаковые имена;
- `document.getElementsByTagName` - возвращает массив элементов, определенных по имени тега. Имя тега является просто именем тега HTML, т.е. 'DIV', 'IMG', 'TABLE', 'A' и т.д.

Существует еще одно свойство, `document.all`, которое выдает массив всех элементов на странице. Однако `document.all` поддерживается не всеми браузерами, поэтому предполагается, что вместо этого используется функция `document.getElementsByTagName("*")`, которая также вернет все элементы на странице.

document.body и document.documentElement

`document.body` ссылается на тег `<BODY>`, где должен предположительно находиться весь контент. Весь DOM сайта вложен в `document.body`. Кроме этого, необходимо использовать `document.body` для определения, что документ был прокручен, и для получения размера окна. К сожалению, это является одной из наиболее сложных вещей, применяемых сегодня в Web-браузерах.

Существует концепция, называемая "Тип документа", которая задает для Web-браузера определенный набор правил. Изменение типа документа заставляет некоторые свойства переместиться из `document.body` в `document.documentElement`, но только некоторые свойства и только для некоторых браузеров.

Проще говоря, это является полным беспорядком, поэтому две следующие функции (будем надеяться) выдадут позицию прокручивания и размеры окна независимо от браузера.

```
function getScrollPos(){
  if (window.pageYOffset){
    return {y:window.pageYOffset, x:window.pageXOffset};
  }
  if(document.documentElement && document.documentElement.scrollTop){
    return {y:document.documentElement.scrollTop,
x:document.documentElement.scrollLeft};
  }
  if(document.body){
    return {y:document.body.scrollTop, x:document.body.scrollLeft};
  }
  return {x:0, y:0};
}

function getWindowDims(){
  if (window.innerWidth){
    return {w:window.innerWidth, h:window.innerHeight};
  }
  if (document.documentElement && document.documentElement.clientWidth){
    return {w:document.documentElement.clientWidth,
h:document.documentElement.clientHeight};
  }
  if (document.body){
    return {w:document.body.clientWidth, h:document.body.clientHeight};
  }
  return {w:0, h:0}
}
```

title, referer

Тремя последними свойствами документа являются title, referer и cookies. document.title и document.referer достаточно понятны. document.title содержит заголовок страницы. Его можно прочитать и изменить после полной загрузки документа. document.referer содержит просто URL страницы, которая привела пользователя на текущую страницу.

Поэтому, если вы щелкнули на ссылке, чтобы попасть на эту страницу, то document.referer будет содержать URL страницы, на которой находится ссылка. Если вы пришли на эту страницу сразу, задавая ее в поле адреса браузера, то document.referer будет неопределен.

Cookie

Последняя тема этой лекции, переменная cookie, отличается от всего остального в JavaScript. cookie является строкой текста, которую можно сохранить с одной страницы на другой, если вы находитесь на одном и том же сервере. В отличие от других переменных в JavaScript, cookie не стирается при перезагрузке страницы. cookie стираются только через определенный период времени или когда все cookie удаляются в браузере.

cookie читают и записывают через document.cookie. В отличие от других свойств изменение document.cookie в действительности не перезаписывает, а добавляет к cookie. Поэтому, если требуется задать 5 cookie, то каждое из них задается с помощью document.cookie= "...";. Формат cookie имеет свои особенности, поэтому мы рассмотрим несколько функций для выполнения этой задачи:

```
function writeCookie(name, value, days){
    if(days){
        (time = new Date()).setTime(new Date().getTime()+days*24*60*60*1000);
        var exp = '; expires='+time.toGMTString();
    }else{
        var exp='';
    }
    document.cookie=name+"="+value+exp+"; path=/";
}

function readCookie(name){
    var cookies = document.cookie.split(';');
    for(var i=0; i<cookies.length; i++){
        var cookie=cookies[i].replace(/^\s+/, '');
        if (cookie.indexOf(name+'=')==0) return cookie.substring(name.length+1);
    }
    return null;
}

function eraseCookie(name){
    writeCookie(name, "", -1);
}
```

Три эти функции выполняют запись, чтение и стирание cookie на текущей странице. Их можно протестировать с помощью следующего кода:

```
function addToCounter() {
    var counter = readCookie('myCounter');
    if(counter){
        counter = parseInt(counter);
    } else {
        counter = 0;
    }

    writeCookie('myCounter', counter+1, 1);
}

function showCounter() {
    alert(readCookie('myCounter'));
}
```

Если увеличить счетчик cookie несколько раз, обновить страницу, а затем проверить счетчик, то можно увидеть, что он остался таким же, как был до обновления страницы. Эти cookie будут сохраняться до тех пор, пока они не будут удалены из браузера или пока не пройдет 24 часа - cookie заданы на период одни сутки.

Литература

<http://www.intuit.ru/department/internet/jsbasics/>

<http://www.intuit.ru/department/internet/js/>

<http://javascript.ru/book?filter0=ru>