Введение в JavaScript

1 ЛЕКЦИЯ: НАЗНАЧЕНИЕ И ПРИМЕНЕНИЕ JAVASCRIPT, ОБЩИЕ СВЕДЕНИЯ

Описано назначение языка JavaScript. Рассмотрены способы внедрения JavaScript-кода в HTML-страницу и принципы его работы.

1.1 Вступление

Гипертекстовая информационная система состоит из множества информационных узлов, множества гипертекстовых связей, определенных на этих узлах и инструментах манипулирования узлами и связями. Технология World Wide Web — это технология ведения гипертекстовых распределенных систем в Internet, и, следовательно, она должна соответствовать общему определению таких систем. Это означает, что все перечисленные выше компоненты гипертекстовой системы должны быть и в Web.

Web как гипертекстовую систему можно рассматривать с двух точек зрения. Во-первых, как совокупность отображаемых страниц, связанных гипертекстовыми переходами (ссылками — контейнер <A>). Во-вторых, как множество элементарных информационных *объектов*, составляющих отображаемые страницы (текст, графика, мобильный код и т.п.). В последнем случае множество гипертекстовых переходов страницы — это такой же информационный фрагмент, как и встроенная в текст картинка.

При втором подходе гипертекстовая сеть определяется на множестве элементарных информационных *объектов* самими HTML-страницами, которые и играют роль гипертекстовых *связей*. Этот подход более продуктивен с точки зрения построения отображаемых страниц "на лету" из готовых компонентов.

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. Если быть более точным, то JavaScript — это не только язык программирования на стороне клиента. Liveware, прародитель JavaScript, является средством подстановок на стороне сервера Netscape. Однако наибольшую популярность JavaScript обеспечило программирование 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 зарегистрированным товарным знаком компании Sun Microsystems. 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. В данный момент JavaScript полностью занимает нишу браузерных языков. На синтаксис JavaScript оказал влияние язык Java, откуда и произошло название JavaScript; как и Java, язык JavaScript является объектным. Однако на этом их связь заканчивается: Java и JavaScript — это разные языки, ни один не является подмножеством другого.

Стандартизация языка была инициирована компанией Netscape и осуществляется ассоциацией ECMA (European Computer Manufacturers Association — Ассоциация европейских производителей компьютеров). Стандартизированная версия имеет название *ECMAScript* и описывается стандартом ECMA-262 (доступна в сети: на английском (http://www.ecma-

international.org/publications/standards/Ecma-262.htm), на русском (http://javascript.ru/ecma)).

Первая версия стандарта (принята в 1997 г.) примерно соответствовала JavaScript 1.1. На данный момент (2008 г) вышла уже третья редакция стандарта (принята в декабре 1999 г), включающая мощные регулярные выражения, улучшенную поддержку строк, новые управляющие конструкции, обработку исключений try/catch, конкретизированное определение ошибок, форматирование при численном выводе и другие изменения. Ведется работа над расширениями и четвертой редакцией стандарта. Отметим, что не все реализации JavaScript на сегодня полностью соответствуют стандарту ECMA. В рамках данного курса мы во всех случаях будем использовать название JavaScript.

1.2 Размещение кода JavaScript 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:

- гипертекстовая ссылка (схема URL);
- обработчик события (в атрибутах, отвечающих событиям);
- подстановка (entity);
- вставка (контейнер <SCRIPT>).

Ниже мы рассмотрим их по очереди. В учебниках по JavaScript описание применения JavaScript обычно начинают с контейнера <SCRIPT>. Но с точки зрения понимания сути взаимодействия JavaScript и HTML это не совсем правильно, поскольку такой порядок не дает ответа на ключевой вопрос: как

JavaScript-код получает управление? Другими словами, 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 же вы имеете только опыт HTML-разметки или давно не писали программ, то тогда лучше начать с программирования гипертекстовых переходов.

Примечание 1. Все последующие примеры Вы можете проверять на работоспособность в Вашем браузере самостоятельно. Для этого скопируйте текст примера в файл (скажем, `primer.htm`); если текст примера состоит только из JavaScript-кода, то заключите его в тэги `<SCRIPT>` и `</SCRIPT>`. Получившийся файл можно просматривать в браузере.

Примечание 2. В данной вводной лекции примеры даются без разбора деталей всех использованных конструкций — воспринимайте их пока интуитивно. Последующие лекции все прояснят. Опишем лишь два важнейших оператора, встречающихся почти в каждом примере. Оператор `alert(строка)` выводит эту строку на экран в *окне предупреждения*, пример такого окна изображен на рис. 3.1. Оператор `document.write(строка)` записывает указанную строку в текущий HTML-документ. Например, следующие два фрагмента HTML-документа равносильны:

Простой HTML-документ	Использование `document.write()`

`<HTML>`
`<BODY>`
`<H1>Заголовок</H1>`
`</BODY>`
`</HTML>`
	`<HTML>`
	`<BODY>`
	`<SCRIPT>`
	`document.write('<H1>Заголовок</H1>');`
	`</SCRIPT>`

	</BODY> </HTML>

1.2.1 Способ 1: 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 mailto. Тип схемы определяется по первому компоненту URL, например:

```
http://its.kpi.ua/directory/page.html
```

В данном случае URL начинается с http — это и есть задание 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, вызываемой при гипертекстовом переходе, является программа загрузки страницы (т.е. при клике по ссылке загружается страница с указанным URL). 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:код_программы">...</A>
<FORM ACTION="JavaScript:код_программы" ...> ... </FORM>
```

В данном случае текст "код_программы" обозначает программу-обработчик на JavaScript, которая вызывается при выборе гипертекстовой ссылки в первом случае и при отправке данных формы (нажатии кнопки

Submit) — во втором. Например, при нажатии на гипертекстовую ссылку "Кликни здесь" можно получить окно предупреждения:

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

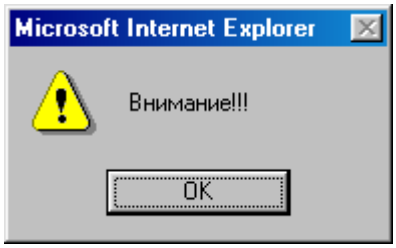


Рисунок 1.1 - Окно предупреждения

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

```
<FORM METHOD=post NAME="form"
  ACTION="JavaScript:form.e.value='Нажали кнопку: Заполнить';void(0);">
<INPUT TYPE=text NAME=e SIZE=30 VALUE=""><BR>
<INPUT TYPE=submit VALUE="Заполнить">
<INPUT TYPE=reset VALUE="Очистить">
</FORM>
```

Пример 1.1. Заполнение поля при нажатии кнопки

В URL можно размещать сложные программы и вызовы функций. Таким образом, при программировании гипертекстового перехода JavaScript-интерпретатор получает управление после того, как пользователь "кликнул" по гипертекстовой ссылке.

1.2.2 Способ 2: обработчики событий

Такие программы, как *обработчики событий*, указываются в атрибутах контейнеров, с которыми эти события связаны. Например, при нажатии на кнопку происходит событие `Click` и соответственно вызывается обработчик этого события `onClick`:

```
<FORM>
<INPUT TYPE=button VALUE="Кнопка"
     onClick="alert('Вы нажали кнопку');">
</FORM>
```

А в момент завершения полной загрузки документа (он связан с контейнером `<BODY>`) происходит событие `Load` и, соответственно, будет вызван обработчик этого события `onLoad`:

```
<BODY onLoad="alert('Приветствуем!');">
```

```
...
</BODY>
```

1.2.3 Способ 3: подстановки

Подстановки (entity) поддерживаются только браузером Netscape Navigator 4.0. Они встречаются на Web-страницах довольно редко. Тем не менее это достаточно мощный инструмент генерации HTML-страницы на стороне браузера. Подстановки имеют формат: `&{код_программы};` и используются в качестве значений атрибутов HTML-контейнеров. В следующем примере поле ввода `INPUT` будет иметь, в качестве значения по умолчанию, адрес текущей страницы, а размер поля будет равным количеству символов в этом адресе.

```
<HTML>
<HEAD>
<SCRIPT>
function l()
{
  str = window.location.href;
  return(str.length);
}
</SCRIPT>
</HEAD>
<BODY>
<FORM><INPUT TYPE=text SIZE="&{l()};"
   VALUE="&{window.location.href};">
</FORM>
</BODY>
</HTML>
```

В случае подстановки JavaScript-интерпретатор получает управление в момент разбора браузером (компонент *парсер*) HTML-документа. Как только парсер встречает конструкцию `&{..};` у атрибута контейнера, он передает управление JavaScript-интерпретатору, который, в свою очередь, после исполнения кода это управление возвращает парсеру. Таким образом, данная операция аналогична подкачке графики на HTML-страницу.

Очевидно, что размещать в заголовке документа генерацию текста страницы бессмысленно — он не будет отображен браузером. Поэтому в заголовок помещают декларации общих переменных и функций, которые будут затем использоваться в теле документа. При этом браузер Netscape Navigator более требовательный, чем Internet Explorer. Если не разместить описание функции в заголовке, то при ее вызове в теле документа можно получить сообщение о том, что данная функция не определена.

В Internet Explorer подстановки не поддерживаются, поэтому пользоваться ими следует аккуратно. Прежде чем выдать браузеру страницу с подстановками, нужно проверить тип этого браузера. Альтернативой подстановкам в Internet Explorer можно считать *динамические свойства* стиля. Например, следующий фрагмент создаст поле ввода, ширина которого в пикселях (px) равна количеству символов в адресе страницы, умноженному на 10:

```
<INPUT TYPE=text style="width:expression(10*location.href.length+'px')">
```

Мы не будем подробно останавливаться на этом способе использования JavaScript-кода.

1.2.4 Способ 4: вставка (контейнер <SCRIPT>)

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 JavaScript-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>.

Помещать JavaScript-код на HTML-странице с помощью контейнера <SCRIPT> можно двумя способами. Первый состоит в написании текста кода непосредственно внутри этого контейнера:

```
<SCRIPT>
 a = 5;
</SCRIPT>
```

Второй способ состоит в том, чтобы вынести код JavaScript в отдельный файл, например, myscript.js (расширение может быть любым), и затем включить его в HTML-страницу следующим образом:

```
<SCRIPT SRC="myscript.js"></SCRIPT>
```

Этот способ удобен, когда один и тот же скрипт планируется использовать на разных HTML-страницах. Обратите внимание, что при наличии атрибута SRC содержимое контейнера <SCRIPT> пусто, и это не случайно: согласно спецификации HTML, если скрипт подключается из внешнего файла, то скрипт, написанный между тэгами <SCRIPT> и </SCRIPT>, если таковой имеется, будет проигнорирован браузером.

Здесь уместно небольшое замечание, которое позволит Вам избежать одной ошибки начинающих программистов. Между тэгами <SCRIPT> и </SCRIPT> не должно встречаться последовательности символов </SCRIPT> в любом контексте. Например, следующий пример работать не будет:

```
<SCRIPT>
alert('</script>');
</SCRIPT>
```

Дело в том, что специфика разбора HTML-документа браузером такова, что он сначала определяет границы скрипта, а потом уже передает его интерпретатору JavaScript. В нашем случае браузер посчитает, что код скрипта завершился на первой же встретившейся ему последовательности символов "</script>", т.е. не на той, на которой было нужно нам. Чтобы пример заработал, достаточно, например, написать alert('<\/script>') (т.к. комбинация "\/" выводит на экран символ "/"), либо разбить строчку на две: alert('</scr'+'ipt>').

Контейнер SCRIPT выполняет две основные функции:

- **размещение кода** внутри HTML-документа;
- **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 в теле документа (внутри контейнера 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 документа размещается внутри контейнера SCRIPT. В следующем примере мы декларировали функцию time_scroll() в заголовке документа, а потом вызвали ее как обработчик события Load в теге начала контейнера BODY.

```
<HTML>
<HEAD>
<SCRIPT>
function time_scroll()
{
  var d = new Date();
  window.status = d.getHours()
          + ':' + d.getMinutes()
          + ':' + d.getSeconds();
  setTimeout('time_scroll()',1000);
}
</SCRIPT>
</HEAD>
<BODY onLoad="time_scroll()">
<H1>Часы в строке статуса</H1>
</BODY>
</HTML>
```

Пример 1.2. Часы в поле статуса окна

Функция `time_scroll()` вызывается по окончании полной загрузки документа (обработчиком `onLoad`). Она заносит текущую дату и время (`new Date`) в переменную `d`. Затем записывает текущее время в формате ЧЧ:ММ:СС в `window.status`, тем самым оно будет отображаться в поле статуса окна браузера (подробнее о нем рассказано в лекции 4). Наконец, она откладывает (`setTimeout`) повторный вызов самой себя на 1000 миллисекунд (т.е. 1 секунду). Таким образом, каждую секунду в поле статуса будет отображаться новое время.

Условная генерация HTML-разметки на стороне браузера

Всегда приятно получать с сервера страницу, подстроенную под возможности нашего браузера или, более того, под пользователя. Существует только две возможности генерации таких страниц: на стороне сервера или непосредственно у клиента. JavaScript-код исполняется на стороне клиента (на самом деле, серверы компании Netscape способны исполнять JavaScript-код и на стороне сервера, только в этом случае он носит название LiveWire-код; не путать с LiveConnect), поэтому рассмотрим только генерацию на стороне клиента.

Для генерации HTML-разметки контейнер `SCRIPT` размещают в теле документа, т.е. внутри контейнера `BODY`. Простой пример — встраивание в страницу локального времени:

```
<BODY>
...
<SCRIPT>
d = new Date();
document.write('Момент загрузки страницы: '
+ d.getHours() + ':'
+ d.getMinutes() + ':'
+ d.getSeconds());
</SCRIPT>
...
</BODY>
```

Пример 1.3. Точное время загрузки страницы

Комментарии в HTML и JavaScript

Несколько слов о различных видах комментариев. В программе JavaScript можно оставлять ***комментарии***, которые игнорируются JavaScript-интерпретатором и служат как пояснения для разработчиков. Однострочные комментарии начинаются с символов `//`. Текст начиная с этих символов и до конца строки считается комментарием. Многострочный комментарий заключается между символами `/*` и `*/` и может простираться на несколько строк.

```
<SCRIPT>

 a=5;  // однострочный комментарий

/* Многострочный
   комментарий     */

</SCRIPT>
```

Для скрытия JavaScript-кода от интерпретации старыми браузерами, не поддерживающими JavaScript (у высокого начальства еще встречаются), весь JavaScript-код между тэгами `<SCRIPT>` и `</SCRIPT>` приходится заключать в HTML-комментарии `<!--` и `-->`. Можно предположить, что эти комбинации символов, не являясь полноценными операторами JavaScript, могут быть неверно поняты JavaScript-интерпретатором и порождать ошибки. Однако этого не происходит, так как разработчики языка ввели соглашение: комбинация символов `<!--` считается началом однострочного комментария (наряду с `//`). Со второй комбинацией (`-->`) такой трюк невозможен (т.к. двойной минус имеет специальное значение в JavaScript), и ее приходится комментировать символами `//`, что иллюстрирует следующий пример.

```
<SCRIPT>
<!-- Скрываем JavaScript-код от старых браузеров

 a = 5;
```

```
// -->
</SCRIPT>
```

Однако в данном курсе мы не будем загромождать примеры такого рода HTML-комментариями, переложив эту обязанность на пользователя. К тому же, все реже можно встретить браузеры, которые вместо выполнения JavaScript-кода выдают его текст в окно браузера.

Указание языка сценария

Контейнер <SCRIPT> имеет необязательный атрибут LANGUAGE, указывающий язык, на котором написан содержащийся внутри контейнера скрипт. Значение атрибута не чувствительно к регистру. Если этот атрибут опущен, то его значением по умолчанию считается "JavaScript". Поэтому все наши примеры можно записывать следующим образом:

```
<SCRIPT LANGUAGE="JavaScript">
...
</SCRIPT>
```

В качестве альтернативы атрибут LANGUAGE может принимать значения "JScript" (упоминавшаяся выше разновидность языка JavaScript, разработанная компанией Microsoft), "VBScript" или "VBS" (оба указывают на язык программирования VBScript, основанный на Visual Basic и тоже являющийся детищем Microsoft; поддерживается преимущественно браузером Internet Explorer) и другие. Кроме того, для JavaScript бывает необходимо указать версию языка, например, LANGUAGE="JavaScript1.2". Потребность в этом может возникнуть, если нужно написать разные участки кода для браузеров, поддерживающих разные версии языка.

Следует также иметь в виду, что в настоящей версии языка HTML (т.е. 4.0 и выше) атрибут LANGUAGE контейнера <SCRIPT> считается устаревшим и нерекомендуемым к использованию (deprecated). Вместо него в контейнере <SCRIPT> рекомендуется использовать атрибут TYPE. Его

значениями, также не чувствительными к регистру, могут быть "text/javascript" (значение по умолчанию), "text/vbscript" и другие. Например, все наши примеры можно оформлять так:

```
<SCRIPT TYPE="text/javascript">
...
</SCRIPT>
```

Некоторые старые браузеры не понимают атрибут TYPE, поэтому можно задавать оба атрибута одновременно — LANGUAGE и TYPE. Атрибут TYPE имеет высший приоритет, т.е. если браузер распознает значение TYPE, то значение LANGUAGE игнорируется.

Поскольку в любом случае значение по умолчанию соответствует языку JavaScript, в наших примерах эти атрибуты будут опускаться.

Регистр символов

Как Вы, наверное, знаете, язык HTML является ***регистро-независимым***. Вследствие этого, контейнер <SCRIPT> можно писать как <script>, его атрибуты — как Type, LANGuage и src, значение атрибутов, указывающих язык, — как "JavaSCRIPT" и "TEXT/JavaScript". Разумеется, значение атрибута SRC, т.е. имя файла, следует писать точно так, как файл назван в операционной системе.

Напротив, язык же JavaScript — ***регистро-зависимый***. Это означает, что все переменные, функции, ключевые слова и т.п. должны набираться в том же регистре, в каком они заданы в языке или в программе пользователя. Например, если Вы объявили переменную var myText='Привет', то в дальнейшем ее можно использовать только как myText, но не MyText. В этом кроется частая ошибка, которую допускают программисты на JavaScript. Она усугубляется еще и тем, что JavaScript не требует явно декларировать переменные, и встретив MyText, интерпретатор может решить, что это новая (но не объявленная) переменная.

Это касается и всех встроенных объектов, свойств и методов языка. Например, объектом является `document`. Вызов `document.write()` нельзя записать как `Document.write()` или `document.Write()`. К свойству объекта `document`, задающему цвет фона Web-страницы, можно обратиться только как `document.bgColor`, а метод этого же объекта, выдающий элемент с заданным идентификатором `"id5"`, можно вызвать только как `document.getElementById("id5")`.

Названия событий, такие как `Click` (щелчок мышью), `DblClick` (двойной щелчок мышью), `Load` (окончание загрузки документа) и т.п. сами по себе не являются элементами синтаксиса. Обработчики же соответствующих событий могут появляться в двух контекстах:

- внутри кода JavaScript — в этом случае регистр имеет значение. Например, чтобы при возникновении события `Load` вызывалась функция `myFunction`, мы должны написать: `window.onload = myFunction`. Названия обработчиков событий `onload`, `onmouseover` и т.п. в таком контексте должны быть написаны маленькими буквами;
- как атрибут какого-либо HTML-контейнера — в этом случае регистр не важен. Например, чтобы обработчик события `onLoad` вызывал функцию `myFunction`, мы можем написать в HTML-исходнике: `<BODY onLoad="myFunction()">` либо `<BODY ONLOAD="myFunction()">`.

2 ЛЕКЦИЯ: ТИПЫ ДАННЫХ И ОПЕРАТОРЫ

Рассматриваются основы синтаксиса языка JavaScript: литералы, переменные, массивы, условные операторы, операторы циклов.

Как и любой другой язык программирования, JavaScript поддерживает встроенные структуры и *типы данных*. Все их многообразие подразделяется на:

- литералы;
- переменные;
- массивы;
- функции;
- объекты.

При этом все они делятся на встроенные и определяемые программистом. Функции и объекты будут рассмотрены в следующей лекции.

2.1 Литералы

Литералом называют данные, которые используются в программе непосредственно. При этом под данными понимаются числа или строки текста. Все они рассматриваются в JavaScript как элементарные *типы данных*. Приведем примеры литералов:

числовой литерал: 10

числовой литерал: 2.310

числовой литерал: 2.3e+2

строковый литерал: 'Это строковый литерал'

строковый литерал: "Это строковый литерал"

Литералы используются в операциях присваивания значений переменным или в операциях сравнения:

```
var a=10;
var str = 'Строка';
if(x=='test') alert(x);
```

Оператор присваивания (`переменная = выражение`) возвращает результат вычисления `выражения`, поэтому ничто не мешает полученное значение присвоить еще и другой переменной. Таким образом, последовательность операторов присваивания выполняется справа налево:

```
result = x = 5+7;
```

Два варианта строковых литералов необходимы для того, чтобы использовать вложенные строковые литералы. Если в строковом литерале требуется использовать одинарную кавычку, то сам литерал можно заключить в двойные кавычки: `"It's cool!"`. Верно и обратное. Но если есть необходимость использовать в строковом литерале оба вида кавычек, то проще всего всех их "экранировать" символом обратной косой черты `\`, при этом саму строку можно заключить в любую пару кавычек. Например:

команда:

```
document.write("It\'s good to say \"Hello\" to someone!");
```

выдаст:

```
It's good to say "Hello" to someone!
```

Помимо строковых литералов (последовательностей символов, заключенных в кавычки) есть еще строковые *объекты*; они создаются конструктором: `var s = new String()`. У этого объекта существует много методов (об объектах и методах пойдет речь в следующей лекции). Следует понимать, что строковый литерал и строковый объект — далеко не одно и то же. Но зачастую мы этого не замечаем, т.к. при применении к строчным литералам методов строчных объектов происходит преобразование первых в последние.

Например, можно сначала присвоить `var s='abra-kadabra'`, а затем применить метод: `var m=s.split('b')`, который неявно преобразует строковый литерал `s` в строковый объект и затем разбивает строку в тех местах, где встречается подстрока `'b'`, возвращая массив строк-кусков: массив `m` будет состоять из строк `'a'`, `'ra-kada'` и `'ra'` (массивы рассматриваются ниже).

2.2 Переменные

Переменная — это область памяти, имеющая свое имя и хранящая некоторые данные. Переменные в JavaScript объявляются с помощью оператора `var`, при этом можно давать или не давать им начальные значения:

```
var k;
var h='Привет!';
```

Можно объявлять сразу несколько переменных в одном операторе `var` (тем самым уменьшая размер кода), но тогда их надо писать через **запятую**. При этом тоже можно давать или не давать начальные значения:

```
var k, h='Привет!';
var t=37, e=2.71828;
```

Тип переменной определяется по присвоенному ей значению. Язык JavaScript — слабо типизирован: в разных частях программы можно присваивать одной и той же переменной значения различных типов, и интерпретатор будет "на лету" менять тип переменной. Узнать тип переменной можно с помощью оператора `typeof()`:

```
var i=5;          alert(typeof(i));
i= new Array();   alert(typeof(i));
i= 3.14;          alert(typeof(i));
i= 'Привет!';     alert(typeof(i));
i= window.open(); alert(typeof(i));
```

Переменная, объявленная оператором `var` вне функций, является ***глобальной*** — она "видна" всюду в скрипте. Переменная, объявленная оператором `var` внутри какой-либо функции, является ***локальной*** — она "видна" только в пределах этой функции. Подробнее о *функциях* будет рассказано в следующем разделе этой лекции.

Например, в следующем фрагменте ничего не будет выведено на экран, несмотря на то, что мы обращаемся к переменной `k` после описания функции, и даже после ее вызова:

```
function f()
{ var k=5; }

f(); alert(k);
```

Причина в том, что переменная k является локальной, и она существует только в процессе работы функции f(), а по окончании ее работы уничтожается.

Если имеется *глобальная* переменная k, а внутри функции объявляется *локальная* переменная с тем же именем (оператором var k), то это будет другая переменная, и изменение ее значения внутри функции никак не повлияет на значение глобальной переменной с тем же именем. Например, этот скрипт выдаст 7, поскольку вызов функции f() не затронет значения глобальной переменной k:

```
var k=7;

function f()
{ var k=5; }

f(); alert(k);
```

То же касается и аргументов при описании функций (с той лишь разницей, что перед ними не нужно ставить var): если имеется *глобальная* переменная k, и мы опишем функцию function f(k) {...}, то переменная k внутри {...} никак не связана с одноименной глобальной переменной. В этом плане JavaScript не отличается от других языков программирования.

Примечание. Объявлять переменные можно и без оператора var, просто присваивая переменной начальное значение. Так зачастую делают с переменными циклов. В следующем примере, даже если переменная i не была объявлена ранее, все будет работать корректно:

```
for(i=0; i<8; i++) { ... }
```

Однако опускать var не рекомендуется. Во-первых, это нарушает ясность кода: если написано i=5, то непонятно, вводится ли здесь новая переменная или меняется значение старой. Во-вторых, и это главное, нужно помнить следующий момент, часто приводящий к неправильной работе программы.

Вне функций объявление переменной без оператора `var` равносильно объявлению с оператором `var` — в обоих случаях переменная будет глобальной. Внутри же функции объявление переменной без оператора `var` делает переменную *глобальной* (а не локальной, как можно было бы предположить), и значит, ее значение могут "видеть" и менять другие функции или операторы вне этой функции. При этом такая переменная становится глобальной не после описания, а после вызова этой функции. Пример:

```
function f()
{ var i=5; k=7; }

f(); alert(k);
```

В приведённом примере после `i=5` стоит точка с запятой (а не запятая). Значит, `k=7` — это отдельное объявление переменной, уже без оператора `var`. Поэтому переменная `k` "видна" снаружи и ее значение (`7`) будет выведено оператором `alert(k)`. Чтобы скрыть переменную `k`, нужно было после `i=5` поставить запятую.

Рассмотрим другой пример, показывающий, к каким неожиданным последствиям может привести отсутствие `var` при описании переменной:

```
function f(i)
{ k=7;
  if(i==3) k=5;
  else { f(3); alert(k); }
}
f(0);
```

Мы вызываем `f(0)`, переменной присваивается значение `k=7`, далее выполнение функции идет по ветке `else` — и оператор `alert(k)` выдает `5` вместо ожидавшегося `7`. Причина в том, что вызов `f(3)` в качестве "побочного эффекта" изменил значение `k`. Чтобы такого не произошло, нужно перед `k=7` поставить `var`. Тогда переменная `k` будет локальной и вызов `f(3)` не сможет ее изменить, так как при вызове функции создаются новые копии всех ее локальных переменных.

При написании больших программ подобные ошибки трудно отследить, поэтому настоятельно рекомендуется все переменные объявлять с оператором `var`, особенно внутри функций.

2.3 Массивы

Массивы делятся на встроенные (`document.links[]`, `document.images[]` и т.п. — их еще называют *коллекциями*) и определяемые пользователем (автором документа). Коллекции будут обсуждаться в следующей лекции. Здесь же мы подробно остановимся на массивах, определяемых пользователем. Для массивов определено несколько методов: `join()`, `reverse()`, `sort()` и другие, а также свойство `length`, которое позволяет получить число элементов массива.

Для определения массива пользователя существует специальный конструктор `Array`. Если ему передается единственный аргумент, причем целое неотрицательное число, то создается незаполненный массив соответствующей длины. Если же передается один аргумент, не являющийся числом, либо более одного аргумента, то создается массив, заполненный этими элементами:

```
a = new Array();
// пустой массив (длины 0)

b = new Array(10);
// массив длины 10

c = new Array(10,'Привет');
// массив из двух элементов: числа и строки

d = [5, 'Тест', 2.71828, 'Число e'];
// краткий способ создать массив из 4 элементов
```

Элементы массива нумеруются с нуля. Поэтому в последнем примере значение `d[0]` равно `5`, а значение `d[1]` равно `'Тест'`. Как видим, массив может состоять из разнородных элементов. Массивы не могут быть

многомерными, однако ничто не мешает завести массив, элементами которого будут тоже массивы.

2.3.1 Метод join()

Метод `join()` позволяет объединить элементы массива в одну строку. Он является обратным к методу `split()`, который разрезает объект типа `String` на куски и составляет из них массив. Кстати, метод `split()` демонстрирует тот факт, что массив можно получить и без конструктора массива.

Рассмотрим пример преобразования локального URL в глобальный URL, где в качестве адреса сервера будет выступать `its.kpi.ua`. Пусть в переменной `localURL` хранится локальный URL некоторого файла:

```
localURL = "file:///D:/courses/internet/js/2/2.html"
```

Разрежем строку в местах вхождения комбинации символов `":/"`, выполнив команду:

```
b = localURL.split(':/')
```

Получим массив:

```
b[0] = "file";
b[1] = "//D";
b[2] = "courses/internet/js/2/2.html";
```

Заменяем 0-й и 1-й элементы на требуемые:

```
b[0] = "http:";
b[1] = "/its.kpi.ua";
```

Наконец, склеиваем полученный массив, вставляя косую черту в местах склейки: `globalURL = b.join("/")`. В итоге мы получаем требуемый глобальный URL — значение `globalURL` будет равно:

```
http://its.kpi.ua/courses/internet/js/2/2.html.
```

2.3.2 Метод reverse()

Метод `reverse()` применяется для изменения порядка элементов массива на противоположный. Предположим, массив упорядочен следующим образом:

```
a = new Array('мать','видит','дочь');
```

Упорядочим его в обратном порядке, вызвав метод

```
a.reverse()
```

Тогда новый массив `a` будет содержать:

```
a[0]='дочь';
a[1]='видит';
a[2]='мать';
```

2.3.3 Метод sort()

Метод `sort()` интерпретирует элементы массива как **строковые литералы** и сортирует массив в **алфавитном** (т.н. лексикографическом) порядке. Обратите внимание: метод `sort()` меняет массив. В предыдущем примере, применив

```
a.sort()
```

мы получим на выходе:

```
a[0]='видит';
a[1]='дочь';
a[2]='мать';
```

Однако, это неудобно, если требуется отсортировать числа, поскольку согласно алфавитному порядку `40` идет раньше чем `5`. Для этих целей у метода `sort()` имеется необязательный аргумент, являющийся именем функции, согласно которой требуется отсортировать массив, т.е. в этом случае вызов метода имеет вид: `a.sort(myfunction)`. Эта функция должна удовлетворять определенным требованиям:

- у нее должно быть ровно два аргумента;
- функция должна возвращать число;

– если первый аргумент функции должен считаться меньшим (большим, равным) чем второй аргумент, то функция должна возвратить отрицательное (положительное, ноль) значение.

Например, если нам требуется сортировать числа, то мы можем описать следующую функцию:

```
function compar(a,b)
{
 if(a < b)  return -1;
 if(a > b)  return  1;
 if(a == b) return  0;
}
```

Теперь, если у нас есть массив

```
b = new Array(10,6,300,25,18);
```

то можно сравнить результаты сортировки без аргумента и с функцией `compar` в качестве аргумента:

```
document.write("Алфавитный порядок:<BR>");
document.write(b.sort());
document.write("<BR>Числовой порядок:<BR>");
document.write(b.sort(compar));
```

В результате выполнения этого кода получим следующее:

```
Алфавитный порядок:
10,18,25,300,6
Числовой порядок:
6,10,18,25,300
```

Обратите внимание: метод `sort()` **интерпретирует** элементы массива как строки (и производит лексикографическую сортировку), но не **преобразует** их в строки. Если в массиве были числа, то они числами и останутся. В этом легко убедиться, если в конце последнего примера выполнить команду `document.write(b[3]+1)`: результат будет `26` (т.е. `25+1`), а не `251` (т.е. `"25"+1`).

2.4 Операторы языка

В этом разделе будут рассмотрены операторы JavaScript. Основное внимание при этом мы уделим операторам декларирования и управления

потоком вычислений. Без них не может быть написана ни одна JavaScript-программа.

Общий перечень этих операторов выглядит следующим образом (сразу оговоримся, что этот список неполный):

- {...}
- if ... else ...
- ()?
- while
- for
- break
- continue
- return

Блок: {...}

Фигурные скобки определяют составной оператор **JavaScript — *блок*.** Основное назначение блока — определение тела цикла, тела условного оператора или функции.

Условный оператор: if ... else ...

Условный оператор применяется для ветвления программы по некоторому логическому условию. Есть два варианта синтаксиса:

```
if (логическое_выражение) оператор;
if (логическое_выражение) оператор_1; else оператор_2;
```

Логическое выражение — это выражение, которое принимает значение `true` или `false`. В первом варианте синтаксиса: если `логическое_выражение` равно `true`, то выполняется указанный `оператор`. Во втором варианте синтаксиса: если `логическое_выражение` равно `true`, то выполняется `оператор_1`, если же оно равно `false` `оператор_2`.

Пример использования (об объекте `navigator` читай лекцию 4):

```
if (navigator.javaEnabled())
 alert('Ваш браузер поддерживает Java');
else
```

```
alert('Ваш браузер НЕ поддерживает Java');
```

Тернарная условная операция: ()?

Этот оператор, называемый *условным выражением*, выдает одно из двух значений в зависимости от выполнения некоторого условия. Синтаксис его таков:

```
(логическое_выражение)? значение_1 : значение_2
```

Если `логическое_выражение` равно `true`, то возвращается `значение_1`, в противном случае `значение_2`. Условное выражение легко имитируется оператором `if...else`, однако оно позволяет сделать более компактным и легко воспринимаемым код программы. Например, следующие два фрагмента равносильны:

```
TheFinalMessage = (k>5)? 'Готово!' : 'Подождите...';
```

```
if(k>5) TheFinalMessage = 'Готово!';
else    TheFinalMessage = 'Подождите...';
```

Оператор цикла while

Оператор `while` задает цикл. Определяется он в общем случае следующим образом:

```
while (условие_продолжения_цикла) тело_цикла;
```

Тело цикла может быть как простым, так и составным оператором. Составной оператор, как всегда, заключается в фигурные скобки. Рекомендуется и простой оператор заключать в них, чтобы программу можно было легко модифицировать. `Условие_продолжения_цикла` является логическим выражением. Тело исполняется до тех пор, пока верно логическое условие. Формально, цикл `while` работает следующим образом:

1) проверяется условие_продолжения_цикла:
    a) если оно ложно (false), цикл закончен,
    b) если же истинно (true), то продолжаем далее;
2) выполняется тело_цикла;
3) переходим к пункту 1.

Такой цикл используется, когда заранее неизвестно количество итераций, например, в ожидании некоторого события. Пример:

```
var s='';
while (s.length<6)
{
 s=prompt('Введите строку длины не менее 6:','');
}
alert('Ваша строка: ' + s + '. Спасибо!');
```

Оператор цикла for

Оператор `for` — это еще один оператор цикла. В общем случае он имеет вид:

```
for (инициализация_переменных_цикла;
     условие_продолжения_цикла;
     модификация_переменных_цикла)  тело_цикла;
```

Тело цикла может быть как простым, так и составным оператором (составной необходимо заключать в фигурные скобки). Операторы `инициализация_переменных_цикла` и `модификация_переменных_цикла` могут состоять из нескольких простых операторов, в этом случае простые операторы должны быть разделены **запятой**. `Условие_продолжения_цикла` является логическим выражением. Цикл `for` работает следующим образом:

1) выполняется инициализация_переменных_цикла;
2) проверяется условие_продолжения_цикла:
    a) если оно ложно (false), цикл закончен,
    b) если же истинно (true), то продолжаем далее;
3) выполняется тело_цикла;
4) выполняется модификация_переменных_цикла;
5) переходим к пункту 2.

Рассмотрим типичный пример использования этого оператора:

```
document.write('Кубы чисел от 1 до 100:');

for (n=1; n<=100; n++)
```

```
 document.write('<BR>'+n+'<sup>3</sup> = '+ Math.pow(n,3));
```

Здесь `Math` — встроенный объект, предоставляющий многочисленные математические константы и функции, а `Math.pow(n,m)` вычисляет степенную функцию n^m.

Оператор выхода из цикла break

Оператор `break` позволяет досрочно покинуть тело цикла. Возвращаясь к нашему примеру с кубами чисел, распечатаем только кубы, не превышающие `5000`.

```
document.write('Кубы чисел, меньшие 5000:');

for (n=1; n<=100; n++)
{
 s=Math.pow(n,3);
 if(s>5000) break;

 document.write('<BR>'+n+'<sup>3</sup> = '+s);
}
```

Несмотря на то, что переменную `n` мы заставили пробегать от `1` до `100`, т.е. заведомо с запасом, реально же цикл выполнится для значений `n` от `1` до ... получите сами!

Оператор перехода к следующей итерации цикла continue

Оператор `continue` позволяет перейти к следующей итерации цикла, пропустив выполнение всех нижестоящих операторов в теле цикла. Если нам нужно вывести кубы чисел от `1` до `100`, превышающие `10 000`, то мы можем составить такой цикл:

```
document.write('Кубы чисел от 1 до 100, большие 10 000:');

for (n=1; n<=100; n++)
{
 s=Math.pow(n,3);
 if(s <= 10000) continue;

 document.write('<BR>'+n+'<sup>3</sup> = '+s);
}
```

Разумеется, для большей гибкости можно использовать в циклах оба оператора `break` и `continue`.

Оператор возврата значения из функции return

Оператор `return` используют для возврата значения из *функции* или обработчика события. Рассмотрим пример с функцией:

```
function sign(n)
{
 if (n>0) return 1;
 if (n<0) return -1;
 return 0;
}

alert( sign(-3) );
```

Обратите внимание: оператор `return` не только указывает, какое значение должна вернуть функция, но и прекращает выполнение дальнейших операторов в теле функции.

При использовании в обработчиках событий оператор `return` позволяет отменить или не отменять действие по умолчанию, которое совершает браузер при возникновении данного события. Отменить его, однако, можно не для всех событий. Рассмотрим пример:

```
<FORM ACTION="newpage.html" METHOD=post>
<INPUT TYPE=submit VALUE="Отправить?"
onClick="alert('Не отправим!');return false;">
</FORM>
```

В этом примере без оператора `return false` пользователь увидел бы окно предупреждения "Не отправим!" и далее был бы перенаправлен на страницу `newpage.html`. Оператор же `return false` позволяет отменить отправку формы, и пользователь лишь увидит окно предупреждения.

Аналогично, чтобы отменить действие по умолчанию для параметров событий `onClick`, `onKeyDown`, `onKeyPress`, `onMouseDown`, `onMouseUp`, `onSubmit`, `onReset`, нужно использовать `return false`.

Для события `onMouseOver` с этой же целью нужно использовать оператор `return true`. Для некоторых же событий, например `onMouseOut`, `onLoad`, `onUnload`, отменить действие по умолчанию невозможно.

3 ЛЕКЦИЯ: ФУНКЦИИ И ОБЪЕКТЫ

Рассматриваются функции как типы данных и как объекты. Рассмотрена в общих чертах объектная модель документа (DOM). Представлены способы описания пользовательских объектов.

Мы объединили описание функций и объектов в одной лекции по причине того, что они тесно взаимосвязаны.

Каждая функция является не только именем для группы операторов, но одновременно и объектом. Объекты же (пользовательские) создаются с помощью функций (конструкторов).

3.1 Функции

Язык программирования не может обойтись без механизма многократного использования кода программы. Такой механизм обеспечивается *процедурами* или *функциями*. В JavaScript *функция* выступает в качестве одного из основных *типов данных*. Одновременно с этим в JavaScript определен класс объектов `Function`.

В общем случае любой *объект* JavaScript определяется через функцию. Для создания *объекта* используется конструктор, который в свою очередь вводится через `Function`. Таким образом, с *функциями* в JavaScript связаны следующие ключевые вопросы:

- функция как тип данных;
- функция как объект;
- функция как конструктор объектов.

Именно эти вопросы мы и рассмотрим в данном разделе.

3.1.1 Функция как тип данных

Определяют *функцию* при помощи ключевого слова `function`:

```
function f(arg1,arg2,...)
```

```
{
/* тело функции */
}
```

Здесь следует обратить внимание на следующие моменты. Во-первых, `function` определяет *переменную* с именем `f`. Эта переменная имеет тип `function`:

```
document.write('Тип переменной f: '+ typeof(f));
// Будет выведено: Тип переменной f: function
```

Во-вторых, эта переменная, как и любая другая, имеет значение — свой исходный текст:

```
var i=5;

function f(a,b,c)
{
 if (a>b) return c;
}

document.write('Значение переменной i: '+ i.valueOf());

// Будет выведено:
// Значение переменной i: 5

document.write('Значение переменной f:<BR>'+ f.valueOf());

// Будет выведено:
// Значение переменной f:
// function f(a,b,c)
// {
//  if (a>b) return c;
// }
```

Как видим, метод `valueOf()` применим как к числовой переменной `i`, так и к переменной `f`, и возвращает их значение. Более того, значение переменной `f` можно присвоить другой переменной, тем самым создав "синоним" функции `f`:

```
function f(a,b,c)
{
 if (a>b) return c;
 else return c+8;
}
```

```
var g = f;
alert('Значение f(2,3,2): '+ f(2,3,2) );
alert('Значение g(2,3,2): '+ g(2,3,2) );

// Будет выведено:
// Значение f(2,3,2): 10
// Значение g(2,3,2): 10
```

Этим приемом удобно пользоваться для сокращения длины кода. Например, если нужно много раз вызвать метод `document.write()`, то можно ввести переменную: `var W = document.write` (обратите внимание — без скобок!), а затем вызывать: `W('<H1>Лекция</H1>')`.

Коль скоро функцию можно присвоить переменной, то ее можно передать и в качестве аргумента другой функции.

```
function kvadrat(a)
{   return a*a;   }

function polinom(a,k)
{ return k(a)+a+5;}

alert(polinom(3,kvadrat));
// Будет выведено: 17
```

Все это усиливается при использовании функции `eval()`, которая в качестве аргумента принимает строку, которую рассматривает как последовательность операторов JavaScript (блок) и выполняет этот блок. В качестве иллюстрации приведем скрипт, который позволяет вычислять функцию `f(f(...f(N)...))`, где число вложений функции `f()` задается пользователем.

```
<SCRIPT>
function kvadrat(a)
{   return a*a;   }

function SuperPower()
{ var
 N = parseInt(document.f.n.value),
 K = parseInt(document.f.k.value),
 L = R = '';
```

```
 for(i=0; i<K; i++)
 {
  L+='kvadrat(';
  R+=')';
 }
 return eval(L+N+R);
}
</SCRIPT>

<FORM NAME=f>
 Введите аргумент (число):
 <INPUT NAME=n><BR>
 Сколько раз возвести его в квадрат?
 <INPUT NAME=k><BR>
 <INPUT TYPE=button value="Возвести" onClick="alert(SuperPower());">
</FORM>
```

Пример 3.1. Многократное вложение функции kvadrat() в себя

Обратите внимание на запись `L=R=''`. Она выполняется справа налево. Сначала происходит присваивание `R=''`. Операция присваивания выдает в качестве результата значение вычисленного выражения (в нашем случае — пустая строка). Она-то и присваивается далее переменной `L`.

Поясним работу скрипта в целом. В функции `SuperPower()` мы сначала считываем значения, введенные в поля формы, и преобразуем их из строк в целые числа функцией `parseInt()`. Далее с помощью цикла `for` мы собираем строку `L`, состоящую из `K` копий строки "`kvadrat(`", и строку `R`, состоящую из `K` правых скобок "`)`". Теперь мы составляем выражение `L+N+R`, представляющее собой `K` раз вложенную в себя функцию `kvadrat()`, примененную к аргументу `N`. Наконец, с помощью функции `eval()` вычисляем полученное выражение. Таким образом, вычисляется функция $(...((N)^2)^2...)^2 = N^{2K}$.

3.1.2 Функция как объект

У любого *типа данных* JavaScript существует *объектовая* "обертка" (wrapper), которая позволяет применять методы типов данных к переменным и литералам, а также получать значения их свойств. Например, длина строки

символов определяется свойством `length`. Аналогичная "обертка" есть и у функций — это класс объектов `Function`.

Например, увидеть значение функции можно не только при помощи метода `valueOf()`, но и используя метод `toString()`:

```
function f(x,y)
{
  return x-y;
}
document.write(f.toString());
```

Результат распечатки:

```
function f(x,y) { return x-y; }
```

Свойства же функции как объекта доступны программисту только тогда, когда они вызываются внутри этой функции. Наиболее часто используемыми свойствами являются: массив (коллекция) аргументов функции (`arguments[]`), его длина (`length`), имя функции, вызвавшей данную функцию (`caller`), и *прототип* (`prototype`).

Рассмотрим пример использования списка аргументов функции и его длины:

```
function my_sort()
{
  a = new Array(my_sort.arguments.length);
  for(i=0;i<my_sort.arguments.length;i++)
     a[i] = my_sort.arguments[i];
  return a.sort();
}

b = my_sort(9,5,7,3,2);
document.write(b);
// Будет выдано: 2,3,5,7,9
```

Чтобы узнать, какая функция вызвала данную функцию, используется свойство `caller`. Возвращаемое ею значение имеет тип `function`. Пример:

```
function s()
{ document.write(s.caller+"<BR>"); }

function M()
{ s(); return 5; }
```

```
function N()
{ s(); return 7; }

M();   N();
```

Результат исполнения:

```
function M() { s(); return 5; }
function N() { s(); return 7; }
```

Еще одним свойством объекта класса `Function` является `prototype`. Но это — общее свойство всех *объектов*, не только функций, поэтому и обсуждать его мы будем в следующем разделе в контексте типа данных `Object`. Упомянем только о конструкторе *объекта* класса `Function`:

```
f = new Function(arg_1,...,arg_n, body)
```

Здесь `f` — это объект класса `Function` (его можно использовать как обычную функцию), `arg_1`, ..., `arg_n` — аргументы функции `f`, а `body` — строка, задающая тело функции `f`.

Данный конструктор можно использовать, например, для описания функций, которые назначают или переопределяют методы объектов. Здесь мы вплотную подошли к вопросу конструирования *объектов*. Дело в том, что переменные внутри функции можно рассматривать в качестве ее свойств, а функции — в качестве методов:

```
function Rectangle(a,b,c,d)
{
this.x0 = a;
this.y0 = b;
this.x1 = c;
this.y1 = d;

this.area = new Function(
 "return Math.abs((this.x1-this.x0)*(this.y1-this.y0))");
}

r = new Rectangle(0,0,30,50);

document.write("Площадь: "+r.area());

// Будет выведено:
// Площадь: 1500
```

Обратите внимание еще на одну особенность — ключевое слово `this`. Оно позволяет сослаться на текущий объект, в рамках которого происходит исполнение JavaScript-кода. В данном случае это объект класса `Rectangle`.

3.2 Объекты

Объект — это главный *тип данных* JavaScript. Любой другой тип данных имеет объектовую "обертку" (wrapper). Это означает, что прежде чем можно будет получить доступ к значению переменной того или иного типа, происходит конвертирование переменной в *объект*, и только после этого выполняются действия над значением. Тип данных `Object` сам определяет объекты.

В сценарии JavaScript могут использоваться объекты нескольких видов:

- ***клиентские объекты***, входящие в модель DOM, т.е. отвечающие тому, что содержится или происходит на Web-странице в окне браузера. Они создаются браузером при разборе (парсинге) HTML-страницы. Примеры: `window`, `document`, `location`, `navigator` и т.п.
- ***серверные объекты***, отвечающие за взаимодействие клиент-сервер. Примеры: `Server`, `Project`, `Client`, `File` и т.п. Серверные объекты в этом курсе рассматриваться не будут.
- ***встроенные объекты***. Они представляют собой различные типы данных, свойства, методы, присущие самому языку JavaScript, независимо от содержимого HTML-страницы. Примеры: встроенные классы объектов `Array`, `String`, `Date`, `Number`, `Function`, `Boolean`, а также встроенный объект `Math`.
- ***пользовательские объекты***. Они создаются программистом в процессе написания сценария с использованием *конструкторов* типа объектов (класса). Например, можно создать свои классы `Cat` и `Dog`. Создание и использование таких объектов будет рассмотрено далее в этой лекции.

3.2.1 Операторы работы с объектами

for ... in ...

Оператор `for(переменная in объект)` позволяет "пробежаться" по свойствам *объекта*. Рассмотрим пример (об объекте `document` см. ниже):

```
for(v in document)
 document.write("document."+v+" = <B>"+ document[v]+"</B><BR>");
```

Результатом работы этого скрипта будет длинный список свойств объекта `document`, мы приведем лишь его начало (полностью получите его самостоятельно):

```
alinkColor = #0000ff
bgColor = #ffffff
mimeType = HTML Document
defaultCharset = windows-1251
lastModified = 07/16/2002 21:22:53
onclick = null
links = [object]
...
```

Примечание Попробуйте запустить этот скрипт в разных браузерах — и Вы увидите, что набор свойств у объекта `document` различный в различных браузерах. Аналогичная ситуация со многими объектами модели DOM, о которой пойдет речь ниже. Именно поэтому приходится постоянно заботиться о так называемой *кроссбраузерной совместимости* при программировании динамических HTML-документов.

with

Оператор `with` задает объект по умолчанию для блока операторов, определенных в его теле. Синтаксис его таков:

```
with (объект) оператор;
```

Все встречающиеся в теле этого оператора свойства и методы должны быть либо записанными полностью, либо они будут считаться свойствами и методами объекта, указанного в операторе `with`. Например, если в документе есть форма с именем `anketa`, а в ней есть поля ввода с именами `age` и

speciality, то мы можем воспользоваться оператором with для сокращения записи:

```
with (document.anketa)
{
 age.value=35;
 speciality.value='программист';
 window.alert(length);
 submit();
}
```

Здесь age.value есть сокращенное обращение к document.anketa.age.value, length есть краткая запись свойства document.anketa.length (означающего число полей в форме), submit() есть краткая запись метода document.anketa.submit() (отсылающего введенные в форму данные на сервер), тогда как метод window.alert() записан полностью и не относится к объекту document.anketa.

Оператором with полезно пользоваться при работе с объектом Math, используемым для доступа к математическим функциям и константам. Например, внутри тела оператора with(Math) можно смело писать: sin(f)*cos(h+PI/2); без оператора with пришлось бы указывать Math три раза: Math.sin(f)*Math.cos(h+Math.PI/2)

3.2.2 Клиентские объекты

Для создания механизма управления страницами на клиентской стороне используется *объектная модель документа* (DOM — Document Object Model). Суть модели в том, что каждому HTML-контейнеру соответствует ***объект***, который характеризуется тройкой:

- *свойства*
- *методы*
- *события*

Объектную модель можно представить как способ связи между страницами и браузером. ***Объектная модель документа*** — это представление *объектов*, их *методов*, *свойств* и *событий*, которые

присутствуют и происходят в программном обеспечении браузера, в виде, удобном для работы с ними из 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.

3.2.3 Иерархия классов DOM

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:

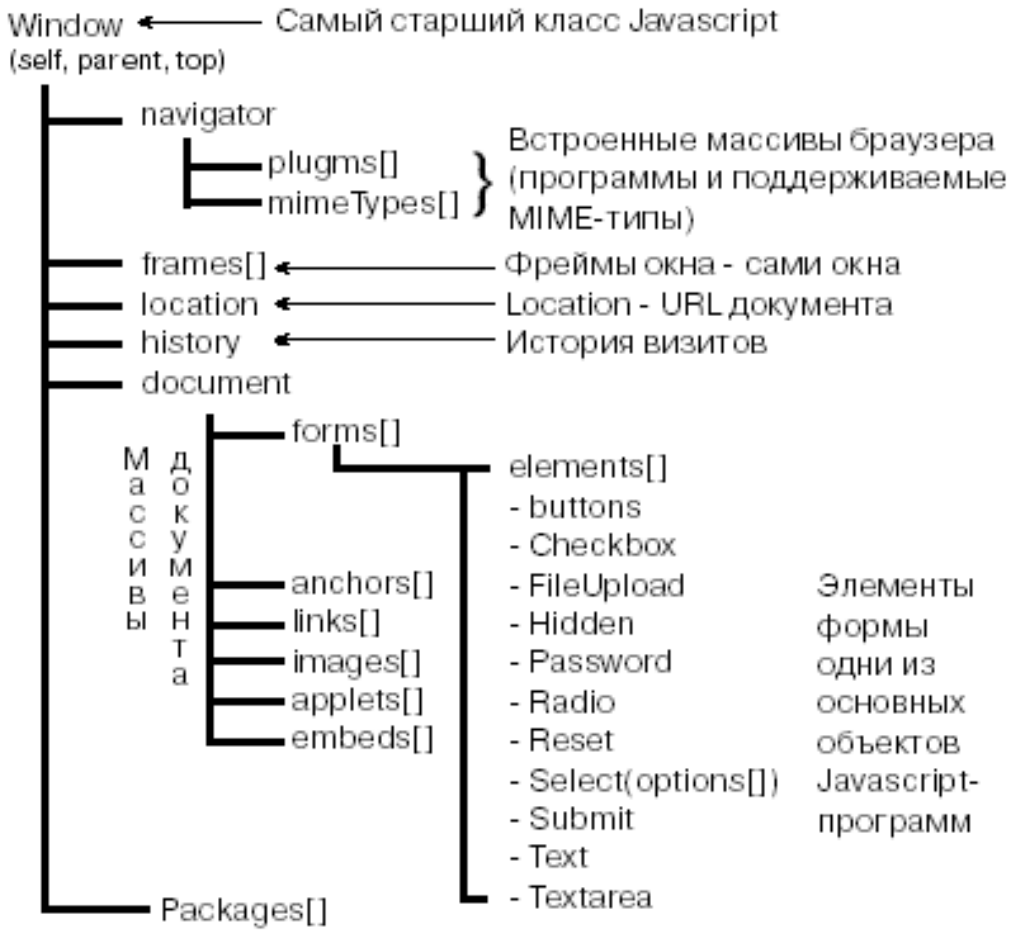


Рисунок 3.1 - Иерархия объектов DOM (фрагмент)

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. Имеется лишь отношение "объект `A` содержит объект `B`" (которое и проиллюстрировано на рис. 3.1). Оно не является иерархией классов в буквальном смысле. Действительно, нахождение класса `window` в этой иерархии выше класса `history` **не означает**, что всякий объект типа `history` является объектом типа `window` и наследует все его свойства и методы, как это понималось бы в стандартных объектно-ориентированных языках. В JavaScript же это отношение означает лишь то, что объект `history` является

свойством объекта `window`, а значит, чтобы получить к нему доступ, нужно воспользоваться "точечной нотацией": `window.history`.

У объектов DOM некоторые свойства обязательно присутствуют, тогда как наличие других зависит от Web-страницы. Например, объект `window` всегда имеет в качестве своих свойств объекты `location` и `history`, т.е. это обязательные свойства. Если HTML-страница содержит контейнер `<BODY>`, то у объекта `window` будет присутствовать в качестве свойства объект `document`. Если HTML-страница содержит контейнер `<FRAMESET>` со вложенными в него контейнерами `<FRAME>`, то у объекта `window` будут присутствовать в качестве свойств имена фреймов, например `window.f1`. Последние, как мы увидим в будущих лекциях, сами являются объектами класса `window`, и для них в свою очередь справедливо все вышесказанное.

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.

3.3 Коллекции

Коллекция — это структура данных JavaScript, похожая на массив. Отличие коллекции от массивов заключается в том, что массивы программист создает сам в коде программы и заполняет их данными; коллекции же создаются браузером и "населяются" объектами, связанными с элементами Web-страницы. Коллекцию можно рассматривать как другой, зачастую более удобный способ доступа к объектам Web-страницы.

Например, если на странице имеются формы с именами `f`, `g5` и `h32`, то у объекта `document` есть соответствующие свойства-объекты `document.f`, `document.g5` и т.д. Но кроме того, у объекта `document` есть свойство `forms`,

являющееся коллекцией (массивом) всех форм, и значит, к тем же самым объектам форм можно обратиться как `document.forms[0]`, `document.forms[1]` и т.д. Это бывает удобным, когда необходимо выполнить какие-то действия со всеми объектами форм на данной странице. Указывая свойства того или иного объекта, мы будем обычно коллекции писать со скобками: `forms[]`, `images[]`, `frames[]`, чтобы подчеркнуть, что это не обычные свойства, а коллекции.

Нумеруются элементы коллекции, начиная с нуля, в порядке их появления в исходном HTML-файле. Доступ к элементам коллекций осуществляется либо по индексу (в круглых или квадратных скобках), либо по имени (тоже в круглых или квадратных скобках, либо через точку), например:

```
window.document.forms[4]     // 5-я форма на странице
window.document.forms(4)     // равносильно предыдущему

window.document.forms['mf']  // форма с именем 'mf'
window.document.forms('mf')  // равносильно предыдущему

window.document.forms.mf     // равносильно предыдущему
window.document.mf           // равносильно предыдущему
```

Способы в 3-4 строчках удобны, когда имя элемента коллекции хранится в качестве значения переменной. Например, если мы задали `var w="mf"`, то мы можем обратиться к форме с именем `"mf"` как `window.document.forms[w]`. Именно так мы поступили выше в разделе про оператор `for...in`, когда выписывали список всех свойств объекта `document`.

Как и у обычных массивов, у коллекций есть свойство `length`, которое позволяет узнать количество элементов в коллекции. Например, `document.images.length`.

Перечислим основные коллекции в объектной модели документа.

Таблица 3.1 - Коллекции в объектной модели документа

Коллекция	Описание
`window.frames[]`	Все фреймы — т.е. объекты, отвечающие

	контейнерам <FRAME>
document.all[]	Все объекты, отвечающие контейнерам внутри контейнера <BODY>
document.anchors[]	Все якоря — т.е. объекты, отвечающие контейнерам <A>
document.applets[]	Все апплеты — т.е. объекты, отвечающие контейнерам <APPLET>
document.embeds[]	Все вложения — т.е. объекты, отвечающие контейнерам <EMBED>
document.forms[]	Все формы — т.е. объекты, отвечающие контейнерам <FORM>
document.images[]	Все картинки — т.е. объекты, отвечающие контейнерам <IMG>
document.links[]	Все ссылки — т.е. объекты, отвечающие контейнерам <A HREF="..."> и <AREA HREF="...">
document.f.elements[]	Все элементы формы с именем f — т.е. объекты, отвечающие контейнерам <INPUT> и <SELECT>
document.f.s.options[]	Все опции (контейнеры <OPTION>) в контейнере <SELECT NAME=s> в форме <FORM NAME=f>
navigator.mimeTypes[]	Все типы MIME, поддерживаемые браузером (список см. на сайте IANA (http://www.iana.org/assignments/media-types/))
function_name.arguments[]	Все аргументы, переданные функции function_name() при вызове

3.4 Свойства

Многие HTML-контейнеры имеют *атрибуты*. Как мы уже знаем, каждому контейнеру соответствует объект. При этом соответствии атрибутам отвечают *свойства* объекта. Соответствие между атрибутами HTML-контейнеров и свойствами DOM-объектов не всегда прямое. Обычно каждому

атрибуту отвечает некоторое свойство объекта. Но, во-первых, название этого свойства не всегда легко угадать по названию атрибута, а во-вторых, у объекта могут быть свойства, не имеющие аналогов среди атрибутов. Кроме того, как мы знаем, атрибуты являются регистро-независимыми, как и весь язык HTML, тогда как свойства объектов нужно писать в точно определенном регистре символов.

Например, контейнер якоря `<A ...>...</A>` имеет атрибут `HREF`, который превращает его в гипертекстовую ссылку:

```
<A HREF="http://its.kpi.ua/">ITS</A>
```

Данной гиперссылке соответствует *объект* (класса `URL`) — `document.links[0]`, если предполагать, что это первая ссылка в нашем документе. Тогда атрибуту `HREF` будет соответствовать свойство `href` этого объекта. К свойству объекта можно обращаться с помощью ***точечной нотации***: `объект.свойство`. Например, чтобы изменить адрес, на который указывает эта ссылка, мы можем написать:

```
document.links[0].href='http://ya.ru/';
```

К свойствам можно также обращаться с помощью ***скобочной нотации***: `объект['свойство']`. В нашем примере:

```
document.links[0]['href']='http://ya.ru/';
```

У объектов, отвечающих гиперссылкам, есть также свойства, не имеющие аналогов среди атрибутов. Например, свойство `document.links[0].protocol` в нашем примере будет равно "http:" и т.д. Полный перечень свойств объектов класса `URL` Вы найдете в лекции 6.

3.5 Методы

В терминологии JavaScript ***методы*** объекта определяют функции, с помощью которых выполняются действия с этим объектом, например, изменение его *свойств*, отображения их на web-странице, отправка данных на сервер, перезагрузка страницы и т.п.

Например, если у нас есть ссылка `<A HREF="http://its.kpi.ua/">ITS</A>` (будем считать, она первая в нашем документе), то у соответствующего ей

объекта `document.links[0]` есть метод `click()`. Его вызов в любом месте JavaScript-программы равносилен тому, как если бы пользователь кликнул по ссылке, что демонстрирует пример:

```
<A HREF="http://its.kpi.ua/">ITS</A>
<SCRIPT> document.links[0].click(); </SCRIPT>
```

При открытии такой страницы пользователь сразу будет перенаправлен на сайт ИТС. Обратите внимание, что скрипт написан после ссылки. Если бы мы написали его до ссылки, то поскольку в этот момент ссылки (а значит и объекта) еще не существует, браузер выдал бы сообщение об ошибке.

Некоторые методы могут применяться неявно. Для всех объектов определен метод преобразования в строку символов: `toString()`. Например, при сложении числа и строки число будет преобразовано в строку:

```
"25"+5 = "25"+(5).toString() = "25"+"5" = "255"
```

Аналогично, если обратиться к объекту `window.location` (рассматриваемом в следующей лекции) в строковом контексте, скажем, внутри вызова `document.write()`, то неявно будет выполнено это преобразование, и программист этого не заметит, как если бы он распечатывал не объект, а строку:

```
<SCRIPT>
document.write('Неявное преобразование: ');
document.write(window.location);
document.write('<BR>Явное преобразование: ');
document.write(window.location.toString());
</SCRIPT>
```

Тот же эффект можно наблюдать для встроенных объектов типа `Date`:

```
<SCRIPT>
var d = new Date();

document.write('Неявное преобразование: ');
document.write(d);
document.write('<BR>Явное преобразование: ');
document.write(d.toString());
</SCRIPT>
```

Результат исполнения получите сами.

3.6 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 этого события — `onClick`. В качестве значения этого атрибута указывается программа обработки события, которую должен написать на JavaScript автор HTML-документа:

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

Обработчики событий указываются в специально созданных для этого атрибутах у тех контейнеров, с которыми эти события связаны. Например, контейнер `BODY` определяет свойства всего документа, поэтому обработчик события "завершена загрузка всего документа" указывается в этом контейнере как значение атрибута `onLoad`.

Примеры событий: нажатие пользователем кнопки в форме, установка фокуса в поле формы или увод фокуса из нее, изменение введенного в поле значения, нажатие кнопки мыши, отпускание кнопки мыши, щелчок кнопкой мыши на объекте (ссылке, поле, кнопке, изображении и т.п.), двойной щелчок кнопкой мыши на объекте, перемещение указателя мыши, выделение текста в поле ввода или на странице и другие. Однако, некоторые изменения, происходящие на странице, не генерируют никаких событий; например: изменение значения в поле ввода не пользователем, а скриптом, изменение фона документа, изменение (скриптом) значения атрибута `HREF` ссылки, а также изменение большинства других атрибутов HTML-контейнеров. Обо всех важных событиях и об их "перехвате" будет рассказываться далее в соответствующих лекциях.

Разные браузеры могут вести себя по-разному при возникновении событий. Рассмотрим следующий пример, позволяющий определить, в каком

порядке вызываются обработчики событий при клике либо двойном клике мыши на ссылке или кнопке:

```
<SCRIPT>
function show_MouseDown() { rrr.innerHTML+='мышь нажали (MouseDown)<br>'; }
function show_Click()     { rrr.innerHTML+='клик мыши (Click)<br>';       }
function show_MouseUp()   { rrr.innerHTML+='мышь отжали (MouseUp)<br>';   }
function show_DblClick()  { rrr.innerHTML+='двойной клик (DblClick)<br>'; }
</SCRIPT>

<A HREF="javascript:void(0);"
   onMouseDown="show_MouseDown();" onClick="show_Click();"
   onMouseUp="show_MouseUp();"  onDblClick="show_DblClick();">Ссылка</A>
<INPUT TYPE=button VALUE="Кнопка"
   onMouseDown="show_MouseDown();" onClick="show_Click();"
   onMouseUp="show_MouseUp();"  onDblClick="show_DblClick();">
<BR><SPAN ID="rrr"></SPAN>
```

Пример 3.2. Слежение за событиями Click и DblClick

Проверьте работу этой странички в Вашем браузере. При одиночном клике на ссылке или кнопке события возникают в порядке: `MouseDown`, `MouseUp`, `Click`, что логично. При двойном же клике последовательность происходящих событий в разных браузерах разная:

в браузере Mozilla Firefox 3.08:

```
MouseDown, MouseUp, Click, MouseDown, MouseUp, Click, DblClick
```

в браузере Internet Explorer 7.0:

```
MouseDown, MouseUp, Click, MouseUp, DblClick
```

Как видим, в Mozilla Firefox последовательность событий более логична — она состоит из двух последовательностей событий, отвечающих одиночному клику, и далее событие двойного клика. Вы можете написать чуть более изощренный скрипт, показывающий, что в IE7 действительно не происходит второго события `Click` при двойном клике мышью.

3.7 Пользовательские объекты

В данном разделе мы остановимся на трех основных моментах:

- понятие объекта;
- прототип объекта;

– методы объекта `Object`.

Мы не будем очень подробно вникать во все эти моменты, так как при программировании на стороне браузера чаще всего обходятся встроенными средствами JavaScript. Но поскольку все эти средства — *объекты*, нам нужно понимать, с чем мы имеем дело.

3.7.1 Понятие пользовательского объекта

Сначала рассмотрим пример определенного пользователем объекта класса `Rectangle`, потом выясним, что же это такое:

```
function Rectangle(a,b,c,d)
{
this.x0 = a;
this.y0 = b;
this.x1 = c;
this.y1 = d;

this.area = new Function(
 "return Math.abs((this.x1-this.x0)*(this.y1-this.y0))");
}

r = new Rectangle(0,0,30,50);
```

Этот же пример использовался выше в разделе "Функции" для иллюстрации применения конструктора `Function`. Здесь мы рассмотрим его в более общем контексте.

Функция `Rectangle()` — это конструктор *объекта* класса `Rectangle`, определенного пользователем. Конструктор позволяет создать экземпляр (объект) данного класса. Ведь *функция* — это не более чем описание некоторых действий. Для того чтобы эти действия были выполнены, необходимо передать функции управление. В нашем примере это делается при помощи оператора `new Rectangle`. Он вызывает функцию `Rectangle()` и тем самым генерирует реальный объект `r`.

В результате этого создается четыре переменных: `x0`, `y0`, `x1`, `y1` — это *свойства* объекта `r`. К ним можно получить доступ только в контексте объекта данного класса, например:

```
up_left_x = r.x0;
up_left_y = r.y0;
```

Кроме свойств, внутри конструктора `Rectangle` мы определили объект `area` класса `Function()`, применив встроенный конструктор языка JavaScript. Это *методы* объекта класса `Rectangle`. Вызвать эту функцию можно тоже только в контексте объекта класса `Rectangle`:

```
sq = r.area();
```

Таким образом, ***объект*** — это совокупность *свойств* и *методов*, доступ к которым можно получить, только создав при помощи конструктора объект данного класса и использовав его контекст.

На практике довольно редко приходится иметь дело с объектами, созданными программистом. Дело в том, что объект создается функцией-конструктором, которая определяется на конкретной странице и, следовательно, все, что создается в рамках данной страницы, не может быть унаследовано другими страницами. Нужны очень веские основания, чтобы автор Web-узла занялся разработкой библиотеки пользовательских классов объектов. Гораздо проще писать функции для каждой страницы.

3.7.2 Прототип

Обычно мы имеем дело со *встроенными объектами* JavaScript, такими как `Data`, `Array` и `String`. Собственно, почти все, что изложено в других разделах курса (кроме иерархии объектов DOM) — это обращение к свойствам и методам встроенных объектов. В этом смысле интересно одно *свойство* объектов, которое носит название `prototype`. *Прототип* — это другое название конструктора объекта конкретного класса. Например, если мы хотим добавить метод к объекту класса `String`, то мы можем это сделать следующим образом:

```
String.prototype.out = new Function("a", "a.write(this)");
```

```
var s = "Привет!";

s.out(document);

// Будет выведено: Привет!
```

Для объявления нового метода для *объектов* класса `String` мы применили конструктор `Function`. Есть один существенный нюанс: новыми методами и свойствами будут обладать только те *объекты*, которые порождаются после изменения прототипа *объекта*. Все встроенные *объекты* создаются до того, как JavaScript-программа получит управление, что существенно ограничивает применение свойства `prototype`.

Тем не менее покажем, как можно добавить метод к встроенному в JavaScript классу. Примером будет служить встроенный поименованный `Image`. Задача состоит в том, чтобы разобрать URL картинки таким же образом, как и URL объекта класса `Link`, т.е. снабдить объект класса `Image` дополнительными методами `protocol()`, `host()` и т.п.:

```
function pr()
{
  a = this.src.split(':');
  return a[0]+':';
}

function ho()
{
  a = this.src.split(':');
  path = a[1].split('/');
  return path[2];
}

function pa()
{
  path = this.src.split('/');
  path[0]='';
  path[2]='';
  return path.join('/').split('///').join('/');
}
```

```
Image.prototype.protocol = pr;
Image.prototype.host = ho;
Image.prototype.pathname = pa;

document.write("<IMG NAME=i1 SRC='image1.gif'><BR>");
document.write(document.i1.src+"<BR>");
document.write(document.i1.protocol()+"<BR>");
document.write(document.i1.host()+"<BR>");
document.write(document.i1.pathname()+"<BR>");
```

Пример 3.3. Добавление методов к классу Image

Как известно, HTML-парсер разбирает HTML-документ и создает встроенные объекты раньше, чем запускается JavaScript-интерпретатор. Поэтому основная идея нашего подхода заключается в том, чтобы переопределить конструктор `Image` раньше, чем он будет использован. Поэтому мы создаем объект `Image` на странице через JavaScript-код. В этом случае сначала происходит переопределение класса `Image`, а уже после этого создается встроенный объект данного класса.

Примечание. При работе с Internet Explorer данный пример работать не будет. Причина в том, что хотя свойство `prototype` имелось в наличии у `String` (см. предыдущий пример), у `Image` такого свойства в данном браузере уже не существует. Однако в Mozilla Firefox все работает корректно.

3.7.3 Методы объекта Object

`Object` — это класс, элементами которого являются любые *объекты* JavaScript. У всех объектов этого класса есть общие методы. Таких методов мы рассмотрим три: `toString()`, `valueOf()` и `assign()`.

Метод `toString()` осуществляет преобразование объекта в строку символов (строковый литерал). Он используется в JavaScript-программах повсеместно, но в основном неявно. Например, при выводе числа или строковых объектов. Интересно применение `toString()` к функциям, например, к функции `pr()` из предыдущего примера:

```
document.write(pr.toString());
```

Результат исполнения:

```
function pr()
{
  a = this.src.split(':');
  return a[0]+':';
}
```

Однако, если распечатать таким же образом объект класса `Image` из того же примера:

```
document.write(document.i1.toString());
```

то получим уже следующее: `[object]` (в Internet Explorer) либо `[object Image]` (в Netscape Navigator). Таким образом, далеко не всегда метод `toString()` возвращает строковый эквивалент содержания объекта.

Аналогично ведет себя и метод `valueOf()`, позволяющий получить значение объекта. В большинстве случаев он работает подобно методу `toString()`, особенно если нужно выводить значение на страницу. Например, оператор `document.write(pr.valueOf())` выдаст то же самое, что и `document.write(pr.toString())` выше.

В отличие от двух предыдущих методов, метод `assign()` позволяет не прочитать, а переназначить какое-либо свойство и метод объекта. Следует заметить, что этот метод работает не во всех браузерах и не со всеми объектами. В общем случае оператор `объект.свойство = значение` равносильно оператору `объект.свойство.assign(значение)`. Например, следующие операторы равносильны — они перенаправляют пользователя на новую страницу:

```
window.location = "http://ya.ru/";
window.location.assign("http://ya.ru/");
```

4 ЛЕКЦИЯ: ПРОГРАММИРУЕМ СВОЙСТВА ОКНА БРАУЗЕРА

Рассматриваются вопросы программирования свойств окна браузера, управление окнами, а также работа с фреймами.

4.1 Объект window

Класс объектов `Window` — это самый старший класс в иерархии объектов JavaScript. Объект `window`, относящийся к текущему окну (т.е. в котором выполняется скрипт), является объектом класса `Window`. Класс объектов `Frame` содержится в классе `Window`, т.е. каждый фрейм — это тоже объект класса `Window`.

О фреймах речь пойдет ниже, а пока вернемся к объекту `window`. Объект `window` создается только в момент открытия окна. Все остальные объекты, которые порождаются при загрузке страницы, есть свойства объекта `window`. Более того, все глобальные переменные, определенные в данном окне, тоже являются свойствами объекта `window`. Таким образом, у объекта `window` могут быть разные свойства при загрузке разных страниц. Кроме того, в разных браузерах свойства объектов и поведение объектов и браузера при обработке событий может быть различным. При программировании на JavaScript чаще всего используют следующие свойства, методы и события объекта `window`:

Таблица 4.1 - Свойства, методы и события объекта ***window***

Свойства	Методы	События

`status`
`defaultStatus`
`location`
`history`
`navigator`
 	
`document`
`frames[]`
 	
`opener`
`parent`
	`open()`
	`close()`
	`focus()`
	`blur()`
	
	`alert()`
	`confirm()`
	`prompt()`
	
	`setTimeout()`
	`setInterval()`
		
		`Load`
		`Unload`
		
		`Focus`
		`Blur`
		
		`Resize`
		`Error`

self top	clearTimeout() clearInterval()	

Поскольку объект window является самым старшим, то в большинстве случаев при обращении к его свойствам и методам приставку "window." можно опускать (разумеется, в случае, если вы хотите обратиться к свойству или методу текущего окна, где работает скрипт; если же это другое окно, то необходимо указать его идентификатор). Так, например, можно писать alert('Привет') вместо window.alert('Привет'), или location вместо window.location. Исключениями из этого правила являются вызовы методов open() и close(), у которых нужно указывать имя окна, с которым работаем (родительское в первом случае и дочернее во втором). Свойства frames[], self, parent и top будут рассмотрены в разделе, посвященном фреймам. Свойство opener будет рассмотрено при описании метода window.close().

4.2 Свойства объекта window

4.2.1 Поле статуса и свойство window.status

Поле статуса — это первое, что начали использовать авторы HTML-страниц из арсенала JavaScript. Калькуляторы, игры, математические вычисления и другие элементы выглядели слишком искусственно. На их фоне бегущая строка в поле статуса была изюминкой, которая могла действительно привлечь внимание пользователей к Web-узлу. Постепенно ее популярность сошла на нет. Бегущие строки стали редкостью, но программирование поля статуса встречается на многих Web-узлах.

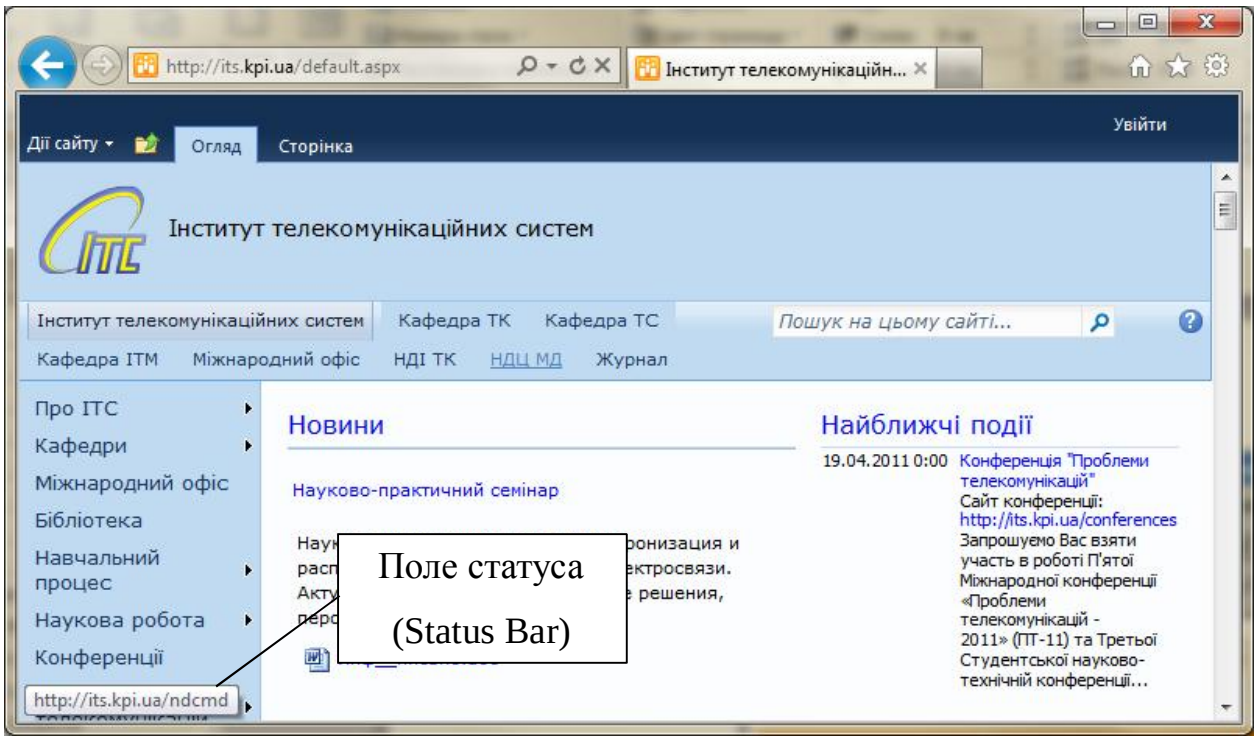


Рисунок 4.1 - Поле статуса

Полем статуса (status bar) называют поле нижней части окна браузера сразу под областью отображения HTML-страницы. В поле статуса отображается информация о состоянии браузера (загрузка документа, загрузка графики, завершение загрузки, запуск апплета и т.п.). Программа на JavaScript имеет возможность работать с этим полем как с изменяемым свойством окна. При этом фактически с ним связаны два разных свойства:

- `window.status` — значение поля статуса;
- `window.defaultStatus` — значение поля статуса по умолчанию.

Значение свойства `status` можно изменить — и оно тут же будет отображено в поле статуса. Свойство `defaultStatus` тоже можно менять — и сразу по его изменении оно отображается в поле статуса.

Разница между этими двумя свойствами заключается в их поведении: если свойству `status` присвоить пустую строку: `window.status=""`, то в поле статуса автоматически будет отображено значение `defaultStatus`. Обратного же не происходит: при присвоении пустой строки свойству `defaultStatus` оно и отобразится в поле статуса, независимо от значения свойства `status`.

Следует отметить, что реакция браузеров на описываемые ниже действия со свойствами `status` и `defaultStatus` может быть разной в различных браузерах.

Программируем status

Свойство `status` связано с отображением сообщений о событиях, отличных от простой загрузки страницы. Например, в Internet Explorer при наведении указателя мыши на ссылку обработчик `onMouseOver` помещает в поле статуса значение URL, указанное в атрибуте `HREF` этой ссылки (при этом никак не меняя значения свойств `status` и `defaultStatus`). При попадании же курсора мыши на область, свободную от ссылок, обработчик `onMouseOut` возвращает в поле статуса значение `defaultStatus`, при условии, что оно не есть пустая строка (опять же никак не меняя значений обоих свойств). Мы можем изменить это поведение, например, как в следующем примере:

```
<A onMouseOver="window.status='Мышь над ссылкой';return true;"
    onMouseOut="window.status='Мышь увели со ссылки';"
   HREF="http://site.com/">Наведите мышь на ссылку и следите за полем статуса</A>
```

Обратите внимание на оператор `return true` в конце обработчика событий `onMouseOver`. Он необходим для того, чтобы *отменить* действие по умолчанию (в данном случае — вывод URL в поле статуса), которое, в отсутствие этого оператора, браузер выполнил бы сразу после вывода нами своей строки в поле статуса, и пользователь не успел бы увидеть нашу строку. Аналогичный трюк отмены действия по умолчанию годится и для некоторых других событий (`onClick`, `onKeyDown`, `onKeyPress`, `onMouseDown`, `onMouseUp`, `onSubmit`, `onReset`), с той лишь разницей, что для перечисленных обработчиков отмена выполняется оператором `return false`.

Для обработчика `onMouseOut` такого способа отменить действие по умолчанию не существует (к сожалению). Но в данном конкретном случае это не требуется — как уже было сказано, при уводе курсора со ссылки в поле статуса восстанавливается значение `defaultStatus` только в случае, если это значение не есть пустая строка. Но в нашем случае (по умолчанию при

загрузке страницы в IE) оно равно именно пустой строке. Поэтому, уводя курсор с нашей ссылки, мы продолжаем видеть в поле статуса строку "Мышь увели со ссылки". Ситуация изменится в следующем примере, когда мы предварительно зададим свое (непустое) значение `defaultStatus`.

Программируем defaultStatus

Свойство `defaultStatus` определяет текст, отображаемый в поле статуса, когда никаких событий не происходит. Дополним предыдущий пример изменением этого свойства в момент окончания загрузки документа, т.е. в обработчике `onLoad`:

```
<BODY onLoad="window.defaultStatus='Значение по умолчанию';">

<A onMouseOver="window.status='Мышь над ссылкой';return true;"
    onMouseOut="window.status='Мышь увели со ссылки'; alert('Ждем');"
   HREF="http://site.com/">Наведите мышь на ссылку и следите за полем статуса</A>

</BODY>
```

Сразу после загрузки документа в поле статуса будет "Значение по умолчанию". При наведении указателя мыши на ссылку в поле статуса появится надпись "Мышь над ссылкой", при этом URL ссылки (`http://site.com/`) в поле статуса не появится, т.к. мы подавили его вывод оператором `return true`.

При убирании указателя мыши со ссылки пользователь бы не успел увидеть строку "Мышь увели со ссылки", поскольку действие по умолчанию (вывод значения `defaultStatus` в поле статуса) не подавлено (и не может быть подавлено — у обработчика `onMouseOut` нет такой возможности). Однако мы ввели оператор вывода окна предупреждения `alert('Ждем')` (он рассматривается ниже) — и теперь пользователь будет видеть в поле статуса строку "Мышь увели со ссылки" до тех пор, пока не нажмет OK на этом окне.

4.2.2 Поле адреса и свойство window.location

Поле адреса в браузере обычно располагается в верхней части окна и отображает URL загруженного документа. Если пользователь хочет вручную перейти к какой-либо странице (набрать ее URL), он делает это в поле адреса.

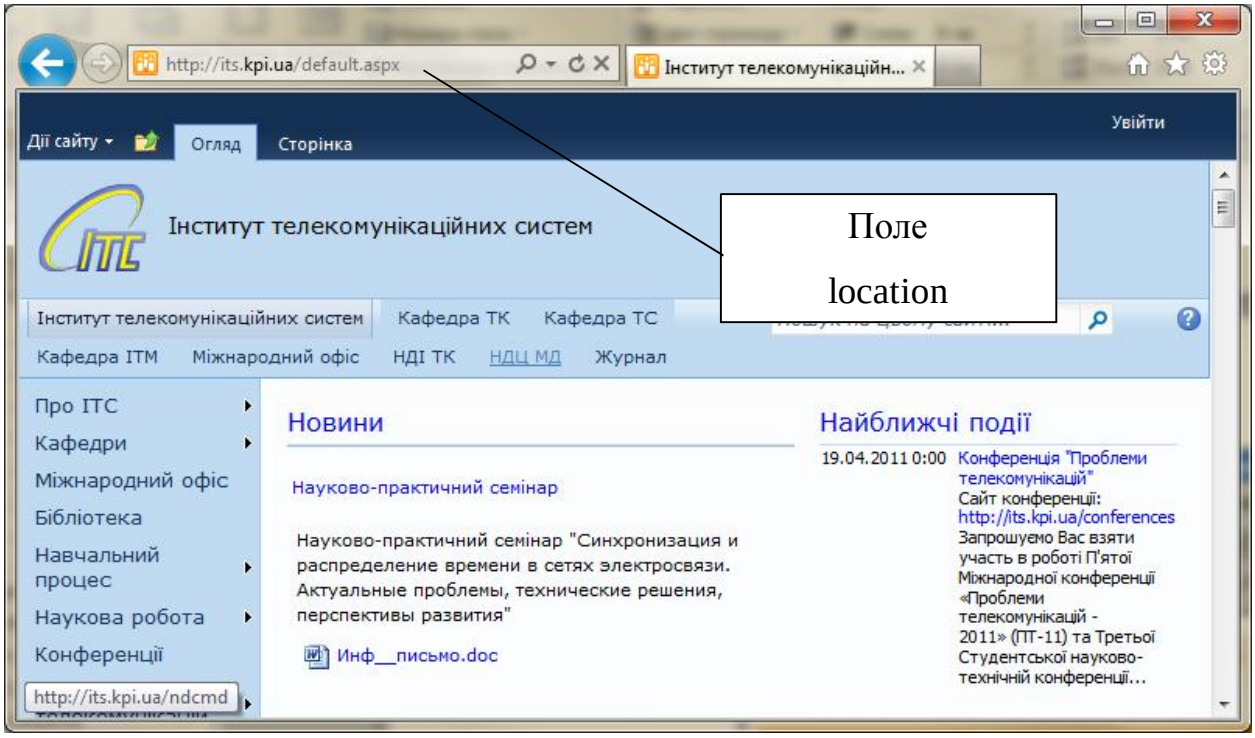


Рисунок 4.2 - Поле адреса (location)

Свойство `location` объекта `window` само является объектом класса `Location`. Класс `Location`, в свою очередь, является подклассом класса `URL`, к которому относятся также объекты классов `Area` и `Link`. Объекты `Location` наследуют все свойства объектов `URL`, что позволяет получить доступ к любой части схемы URL. Подробнее о классе объектов `URL` мы расскажем в лекции 6.

В целях совместимости с прежними версиями JavaScript, в языке поддерживается также свойство `window.document.location`, которое в настоящее время полностью дублирует свойство `window.location` со всеми его свойствами и методами. Рассмотрим теперь свойства и методы объекта `window.location` (событий, связанных с этим объектом, нет).

Свойства объекта location

Их проще продемонстрировать на примере. Предположим, что браузер отображает страницу, расположенную по адресу:

```
http://www.site.ru:80/dir/page.cgi?product=phone&id=3#mark
```

Тогда свойства объекта `location` примут следующие значения:

```
window.location.href                                                =
"http://www.site.ru:80/dir/page.cgi?product=phone&id=3#mark"
window.location.protocol = "http:"
window.location.hostname = "www.site.ru"
window.location.port     = 80
window.location.host     = "www.site.ru:80"
window.location.pathname = "dir/page.cgi"
window.location.search   = "?product=phone&id=3"
window.location.hash     = "#mark"
```

Как уже говорилось в предыдущих лекциях, к свойствам объектов можно обращаться как с помощью точечной нотации (как выше), так и с помощью скобочной нотации, например: `window.location['host']`.

Методы объекта location

Методы объекта `location` предназначены для управления загрузкой и перезагрузкой страницы. Это управление заключается в том, что можно либо перезагрузить текущий документ (метод `reload()`), либо загрузить новый (метод `replace()`).

```
window.location.reload(true);
```

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 кнопки Reload браузера (клавиши F5 в Internet Explorer). Если в качестве аргумента указать `false`, то браузер перезагрузит текущий документ с сервера, несмотря ни на что. Такое поведение соответствует

одновременному нажатию клавиши Shift и кнопки браузера Reload (или Ctrl+F5 в Internet Explorer).

Используя объект `location`, перейти на новую страницу можно двумя способами:

```
window.location.href="http://www.newsite.ru/";
window.location.replace("http://www.newsite.ru/");
```

Разница между ними — в отображении этого действия в истории посещений страниц `window.history`. В первом случае в историю посещений добавится новый элемент, содержащий адрес "`http://www.newsite.ru/`", так что при желании можно будет нажать кнопку Back на панели браузера, чтобы вернуться к прежней странице. Во втором случае новый адрес "`http://www.newsite.ru/`" заместит прежний в истории посещений, и вернуться к прежней странице нажатием кнопки Back уже будет невозможно.

4.2.3 История посещений (history)

История посещений страниц World Wide Web позволяет пользователю вернуться к странице, которую он просматривал ранее в данном окне браузера. История посещений в JavaScript трансформируется в объект `window.history`. Этот объект указывает на массив URL-страниц, которые пользователь посещал и которые он может получить, выбрав из меню браузера режим Go. Методы объекта `history` позволяют загружать страницы, используя URL из этого массива.

Чтобы не возникло проблем с безопасностью браузера, путешествовать по History можно, только используя индекс. 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 вернутся 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. Аналогичным образом действует метод `history.forward()`, перенося нас на следующую посещенную страницу.

Существует также метод `go()`, имеющий целочисленный аргумент и позволяющий перескакивать на несколько шагов вперед или назад по истории посещений. Например, `history.go(-3)` перенесет нас на 3 шага назад в истории просмотра. При этом методы `back()` и `forward()` равносильны методу `go()` с аргументами `-1` и `1`, соответственно. Вызов `history.go(0)` приведет к перезагрузке текущей страницы.

4.2.4 Тип браузера (navigator)

Часто возникает задача настройки страницы на конкретную программу просмотра (браузер). При этом возможны два варианта: определение типа браузера на стороне сервера, либо на стороне клиента. Для последнего варианта в арсенале объектов JavaScript существует объект `window.navigator`. Важнейшие из свойств этого объекта перечислены ниже.

Таблица 4.2 - Основные свойства объекта ***`window.navigator`***

Свойство	Описание
`userAgent`	Основная информация о браузере. Передается серверу в HTTP-заголовке при открытии пользователем страниц
`appName`	Название браузера
`appCodeName`	Кодовое название браузера
`appVersion`	Данные о версии браузера и совместимости

Рассмотрим простой пример определения типа программы просмотра:

```
<FORM><INPUT TYPE=button VALUE="Тип навигатора"
onClick="alert(window.navigator.userAgent);"></FORM>
```

При нажатии на кнопку отображается окно предупреждения, содержащее значение свойства `navigator.userAgent`. Если это значение разобрать по компонентам, то может получиться, например, следующее:

```
navigator.userAgent    = "Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.0;
SLCC1)"

navigator.appName     = "Microsoft Internet Explorer"
navigator.appCodeName = "Mozilla"
navigator.appVersion  = "4.0 (compatible; MSIE 7.0; Windows NT 6.0; SLCC1)"
```

У объекта `navigator` есть еще несколько интересных с точки зрения программирования применений. Например, чтобы проверить, поддерживает ли браузер клиента язык Java, достаточно вызвать метод `navigator.javaEnabled()`, возвращающий значение `true`, если поддерживает, и `false` в противном случае.

Можно проверить, какие форматы графических файлов поддерживает браузер, воспользовавшись свойством `navigator.mimeTypes` (оно представляет собой массив всех типов MIME, которые поддерживаются данным браузером):

```
<SCRIPT>
if(navigator.mimeTypes['image/gif']!=null)
  document.write('Ваш браузер поддерживает GIF<BR>');
if(navigator.mimeTypes['image/tif']==null)
  document.write('Ваш браузер не поддерживает TIFF');
</SCRIPT>
```

К сожалению, такая проверка не позволяет определить наличие возможности автоматической подгрузки графики.

4.3 Методы объекта window

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. Сосредоточимся на простых и наиболее популярных методах управления окнами.

4.3.1 alert()

Метод `alert()` позволяет выдать *окно предупреждения*, имеющее единственную кнопку "OK":

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

Нужно лишь иметь в виду, что сообщения выводятся системным шрифтом, следовательно, для получения предупреждений на русском языке нужна локализованная версия ОС.

4.3.2 confirm()

Метод `confirm()` позволяет задать пользователю вопрос, на который тот может ответить либо положительно (нажав кнопку "OK"), либо отрицательно (нажав кнопку "Отмена" или "Cancel", либо просто закрыв окно запроса). В соответствии с действиями пользователя метод `confirm()` возвращает значение `true` либо `false`. Пример:

```
<FORM NAME=f>
<INPUT TYPE=button NAME=b VALUE="Нажмите эту кнопку"
onClick="if(window.confirm('Вы знаете JavaScript?'))
     document.f.b.value='Да. Спросить еще?';
else document.f.b.value='Нет. Спросить еще?';">
</FORM>
```

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

4.3.3 prompt()

Метод `prompt()` позволяет принять от пользователя строку текста. Синтаксис его таков:

```
prompt("Строка вопроса","Строка ответа по умолчанию")
```

Когда пользователь введет свой ответ (либо оставит неизменным ответ по умолчанию) и нажмет кнопку OK, метод `prompt()` возвратит полученную строчку в качестве значения, которое можно далее присвоить любой переменной и потом разбирать ее в JavaScript-программе.

```
<FORM NAME=f>
<INPUT TYPE=button VALUE="Открыть окно ввода"
onClick="document.f.e.value=
     window.prompt('Введите сообщение','Сюда');">
<INPUT SIZE=30 NAME=e>
</FORM>
```

4.3.4 window.open()

Метод ***open()*** предназначен для создания новых *окон*. В общем случае его синтаксис выглядит следующим образом:

```
myWin                                                    =
window.open("URL","имя_окна","параметр=значение,параметр=значение,...",
заменить);
```

Первый аргумент задает адрес страницы, загружаемой в новое окно (можно оставить пустую строку, тогда окно останется пустым). Второй аргумент задает имя окна, которое можно будет использовать в атрибуте `TARGET` контейнеров `<A>` и `<FORM>`. В качестве значений допустимы также зарезервированные имена `_blank`, `_parent`, `_self`, `_top`, смысл которых такой же, как у аналогичных значений атрибута `TARGET`. Если `имя_окна` совпадает с именем уже существующего окна (или фрейма), то новое окно не создается, а все последующие манипуляции с переменной `myWin` будут применяться к этому окну (или фрейму).

Третий аргумент есть **не содержащая пробелов** строка, представляющая собой список параметров и их значений, перечисленных через запятую. Указание каждого из параметров необязательно, однако значения по умолчанию могут зависеть от браузера, поэтому всегда указывайте явно те параметры, на которые рассчитываете. Возможные параметры перечислены в таблице 4.3. Вместо значений `yes` и `no` можно использовать `1` и `0`. Последний аргумент "`заменить`" является необязательным, принимает значения `true` и `false` и означает: следует ли новый URL добавить в `history` в качестве нового элемента или заменить им последний элемент `history`.

Метод `window.open()` возвращает ссылку на вновь открытое окно, т.е. объект класса `Window`. Его можно присвоить переменной (что мы и сделали выше), с тем чтобы потом можно было управлять открытым окном (писать в него, читать из него, передавать и убирать фокус, закрывать).

Таблица 4.3 - Параметры метода ***`window.open()`***

Параметр	Значения	Описание
`width`	число	Ширина окна в пикселах (не менее `100`)
`height`	число	Высота окна в пикселах (не менее `100`)
`left`	число	Расстояние от левого края экрана до левой границы окна в пикселах
`top`	число	Расстояние от верхнего края экрана до верхней границы окна в пикселах
`directories`	`yes/no`	Наличие у окна панели папок (Netscape Navigator)
`location`	`yes/no`	Наличие у окна поля адреса
`menubar`	`yes/no`	Наличие у окна панели меню
`resizable`	`yes/no`	Сможет ли пользователь менять размер окна
`scrollbars`	`yes/no`	Наличие у окна полос прокрутки
`status`	`yes/no`	Наличие у окна поля статуса
`toolbar`	`yes/no`	Наличие у окна панели инструментов

Приведем два примера открытия нового окна:

```
<FORM>
<INPUT TYPE=button VALUE="Простое окно"
 onClick="window.open('', '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('', 'test2',
  'directories=yes,height=200,location=yes,'+
  'menubar=yes,resizable=yes,scrollbars=yes,'+
  'status=yes,toolbar=yes,width=200');">
</FORM>
```

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

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

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

- `directories=yes` — окно с панелью папок
- `height=200` — высота 200 px
- `location=yes` — поле адреса 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 px

4.3.5 window.close()

Метод ***`close()`*** позволяет закрыть окно. Чаще всего возникает вопрос, какое из окон, собственно, следует закрыть. Если необходимо закрыть текущее, то:

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

Если мы открыли окно с помощью метода `window.open()`, то из скрипта, работающего в новом окне, сослаться на окно-родитель можно с помощью

window.opener (обратите внимание, здесь window ссылается на объект нового, созданного окна, т.к. оно использовано в скрипте, работающем в новом окне). Поэтому, 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.

4.3.6 Методы focus() и blur()

Метод focus() применяется для передачи фокуса в окно, с которым он использовался. Передача фокуса полезна как при открытии окна, так и при его закрытии, не говоря уже о случаях, когда нужно выбирать окна. Рассмотрим пример.

Открываем окно и, не закрывая его, снова откроем окно с таким же именем, но с другим текстом. Новое окно не появилось поверх основного окна, так как фокус ему не был передан. Теперь повторим открытие окна, но уже с передачей фокуса:

```
<HTML>
<HEAD>
<SCRIPT>
function myfocus(a)
{
  myWin = window.open('','example','width=300,height=200');
  // открываем окно и заводим переменную с указателем на него.
  // Если окно с именем 'example' существует, то новое окно не создается,
  // а открывается поток для записи в имеющееся окно с именем 'example'

  if(a==1)
  {
```

```
    myWin.document.open(); //открываем поток ввода в уже созданное окно
    myWin.document.write('<H1>Открыли окно в первый раз'); //Пишем в этот
поток
  }

  if(a==2)
  {
    myWin.document.open();
    myWin.document.write('<H1>Открыли окно во второй раз');
  }

  if(a==3)
  {
    myWin.focus(); // передаем фокус, а затем выполняем те же действия,
                // что и в предыдущем случае
    myWin.document.open();
    myWin.document.write('<H1>Открыли окно в третий раз');
  }

  myWin.document.write('</H1>');
  myWin.document.close();
}
</SCRIPT>
</HEAD>
<BODY>
<a href="javascript:myfocus(1);">Откроем окно и напишем в него что-то</a>,
<BR><BR>
<a href="javascript:myfocus(2);">напишем в него же что-то другое, но фокус не
передадим</a>,
<BR><BR>
<a  href="javascript:myfocus(3);">опять что-то напишем в него, но сперва
передав ему фокус</a>.
</BODY>
</HTML>
```

Пример 4.1. Передача фокуса в новое окно

Поскольку мы пишем содержание нового окна из окна старого (родителя), то в качестве указателя на объект используем значение переменной `myWin`.

Чтобы увести фокус из определенного окна `myWin`, необходимо применить метод `myWin.blur()`. Например, чтобы увести фокус с текущего

окна, где выполняется скрипт, нужно вызвать `window.blur()`. Эффект будет тот же, как если бы пользователь сам свернул окно нажатием кнопки  в правом верхнем углу окна.

4.3.7 Метод setTimeout()

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

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

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

```
<HTML><HEAD><SCRIPT>
var Chasy_idut=false;

function myclock()
{
  if(Chasy_idut)
  {
    d = new Date();
    document.f.c.value =
    d.getHours()+':'+
    d.getMinutes()+':'+
    d.getSeconds();
  }
 setTimeout("myclock();",500);
}

function FlipFlag()
{
  Chasy_idut = !Chasy_idut;
  document.f.b.value = (Chasy_idut)?
   'Остановить' : 'Запустить';
}
</SCRIPT></HEAD>
<BODY onLoad="myclock();">
<FORM NAME=f>
Текущее время:<INPUT NAME=c size=8>
<INPUT TYPE=button name=b VALUE="Запустить"
```

```
onClick="FlipFlag();">
</FORM></BODY></HTML>
```

Пример 4.2. Часы с использованием setTimeout()

Обратите внимание, что поток порождается (т.е. вызывается `setTimeout()`) всегда, даже в том случае, когда мы остановили показ часов. Если бы он создавался только при значении переменной `Chasy_idut = true`, то часы бы просто не запустились, так как в самом начале исполнения скрипта мы установили `var Chasy_idut = false`. Но даже если бы мы установили в начале `var Chasy_idut = true`, то часы бы запустились при загрузке страницы, а после остановки поток бы исчез, и при последующем нажатии кнопки "Запустить" часы продолжали бы стоять.

4.3.8 Метод 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:

```
<HTML><HEAD><SCRIPT>
var Chasy_idut=false;
var potok;

function StartClock()
{
  d = new Date();
  document.f.c.value =
   d.getHours()+':'+
   d.getMinutes()+':'+
   d.getSeconds();
  potok = setTimeout('StartClock();',500);
  Chasy_idut=true;
}

function StopClock()
{
```

```
  clearTimeout(potok);
  Chasy_idut=false;
}
</SCRIPT></HEAD><BODY>
<FORM NAME=f>
Текущее время:<INPUT NAME=c size=8>
<INPUT TYPE=button VALUE="Запустить" onClick="if(!Chasy_idut) StartClock();">
<INPUT TYPE=button VALUE="Остановить" onClick="if(Chasy_idut) StopClock();">
</FORM></BODY></HTML>
```

Пример 4.3. Часы с использованием setTimeout() и clearTimeout()

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

4.3.9 Методы setInterval() и clearInterval()

В предыдущих примерах для того, чтобы поток запускался снова и снова, мы помещали в функцию в качестве последнего оператора вызов метода `setTimeout()`. Однако в JavaScript для этих целей имеются специальные методы. Метод `setInterval("код_JavaScript",time)` выполняет `код_JavaScript` с периодом раз в `time` миллисекунд. Возвращаемое значение — ссылка на созданный поток. Чтобы остановить поток, необходимо вызвать метод `clearInterval(поток)`.

4.4 События объекта window

Остановимся вкратце на событиях, связанных с объектом `window`. Обработчики этих событий обычно помещают как атрибут контейнера `<BODY>`.

- `Load` — событие происходит в момент, когда загрузка документа в данном окне полностью закончилась. Если текущим окном является фрейм, то событие `Load` его объекта `window` происходит, когда в данном фрейме загрузка документа закончилась, независимо от состояния загрузки документов в других фреймах.

Использовать обработчик данного события можно, например, следующим образом:

```
<BODY onLoad="alert('Документ полностью загружен.');">
```

- `Unload` — событие происходит в момент выгрузки страницы из окна. Например, когда пользователь закрывает окно, либо переходит с данной Web-страницы на другую, кликнув ссылку или набрав адрес в адресной строке, либо при изменении адреса страницы (свойства `window.location`) скриптом. Например, при уходе пользователя с нашей страницы мы можем позаботиться о его удобстве и закрыть открытое ранее нашим скриптом окно:

```
<BODY onUnload="myWin.close();">
```

- `Error` — событие происходит при возникновении ошибки в процессе загрузки страницы. Если это событие произошло, можно, например, вывести сообщение пользователю с помощью `alert()` или попытаться перезагрузить страницу с помощью `window.location.reload()`. В следующем примере мы назначаем обработчиком события `Error` функцию `ff()`, которая будет выдавать сообщение. В тексте программы мы допустили ошибку: слово `Alert` написано с заглавной буквы (помните, что в JavaScript это недопустимо?). Поэтому при открытии этого примера возникнет ошибка и пользователь получит об этом "дружественное" сообщение.

```
<SCRIPT>
function ff()
{ alert('Произошла ошибка. Свяжитесь с Web-мастером.'); }

window.onerror = ff;

Alert('Привет');
</SCRIPT>
```

- `Focus` — событие происходит в момент, когда окну передается фокус. Например, когда пользователь "раскрывает" свернутое ранее окно, либо (в Windows) выбирает это окно браузера с

помощью Alt+Tab среди окон других приложений. Это событие происходит также при программной передаче фокуса данному окну путем вызова метода `window.focus()`. Пример использования:

```
<BODY onFocus="alert('Спасибо, что снова вернулись!');">
```

- `Blur` — событие, противоположное предыдущему, происходит в момент, когда данное окно теряет фокус. Это может произойти в результате действий пользователя либо программными средствами — вызовом метода `window.blur()`.
- `Resize` — событие происходит при изменении размеров окна пользователем либо сценарием.

4.5 Переменные как свойства окна

Глобальные переменные на самом деле являются свойствами объекта `window`. В следующем примере мы открываем окно с идентификатором `wid`, заводим в нем глобальную переменную `t` и затем пользуемся ею в окне-родителе, ссылаясь на нее как `wid.t`:

```
<HTML><HEAD>
<SCRIPT>
wid = window.open('','','width=750,height=100,status=yes');
wid.document.open(); R = wid.document.write;
R('<HTML><HEAD><SCRIPT>var t;<\/SCRIPT></HEAD>');
R('<BODY><H1>Новое окно</H1></BODY></HTML>');
wid.document.close();
</SCRIPT>
</HEAD>
<BODY>
<A HREF="javascript:
wid.t=window.prompt('Новое состояние:','');
wid.status=wid.t; wid.focus(); void(0);"
>Изменим значение переменной t в новом окне</A>
</BODY></HTML>
```

Пример 4.4. Изменение переменной открытого окна

Обратите внимание на нюанс: внутри скрипта мы написали `<\/SCRIPT>`. Комбинация "`\/`" выдает на выходе "`/`". Сделали мы это для того, чтобы браузер (точнее, его HTML-парсер) не воспринял бы `</SCRIPT>` как

завершающий тэг нашего (внешнего) скрипта. Подробнее этот аспект обсуждался во вводной лекции. Также обратите внимание на алиас (синоним) `R`, который мы дали методу `wid.document.write`, чтобы иметь возможность кратко вызывать его как `R(...)`.

Аналогичным образом (с приставкой `wid`, указывающей на объект окна) можно обращаться ко всем элементам, находящимся в открытом нами окне, например, к формам. В качестве примера рассмотрим изменение поля ввода в окне-потомке из окна-предка. Создадим дочернее окно с помощью функции `okno()`, в нем создадим форму, а затем обратимся к полю этой формы из окна-предка:

```
<HTML>
<HEAD>
<SCRIPT>
var wid; // Объявляем глобальную переменную
function okno()
{
  wid = window.open('','okoshko','width=500,height=200');
  wid.document.open(); R = wid.document.write;
  R('<HTML><BODY><H1>Меняем текст в окне-потомке:</H1>');
  R('<FORM NAME=f><INPUT SIZE=40 NAME=t VALUE=Текст>');
  R('</FORM></BODY></HTML>');
  wid.document.close();
}
</SCRIPT>
</HEAD>
<BODY>
<INPUT TYPE=button VALUE="Открыть окно примера" onClick="okno()">
<INPUT TYPE=button VALUE="Написать текущее время в поле ввода"
       onClick="window.wid.document.f.t.value=new Date();
                window.wid.focus();">
</BODY>
</HTML>
```

Пример 4.5. Изменение поля статуса в открытом окне

Открывая окно-потомок, мы поместили в переменную `wid` указатель на окно: `wid=window.open(...);`. Теперь мы можем использовать `wid` как идентификатор *объекта* класса `Window`. Вызов метода `window.wid.focus()` в нашем случае обязателен, поскольку при нажатии на кнопку "Написать

текущее время в поле ввода" происходит передача фокуса в родительское окно (которое может заслонять вновь открытое окно, так что изменения, происходящие в окне-потомке, не будут видны пользователю). Для того, чтобы увидеть изменения, мы должны передать фокус в окно-потомок.

Переменная `wid` должна быть *глобальной*, т.е. определена за пределами каких-либо функций (как сделано в нашем примере). В этом случае она становится *свойством* объекта `window`, поэтому мы обращаемся к ней в обработчике `onClick` посредством `window.wid`. Если бы мы поместили ее внутри описания функции `okno()`, написав `var wid = window.open(...)`, то мы не смогли бы к ней обратиться из обработчика события `onClick`, находящегося вне функции `okno()`.

4.6 Объект document

Объект `document` является важнейшим свойством объекта `window` (т.е. полностью к нему нужно обращаться как `window.document`). Все элементы HTML-разметки, присутствующие на web-странице, — текст, абзацы, гиперссылки, картинки, списки, таблицы, формы и т.д. — являются свойствами объекта `document`. Можно сказать, что технология DHTML (Dynamic HTML), т.е. динамическое изменение содержимого web-страницы, заключается именно в работе со свойствами, методами и событиями объекта `document` (не считая работы с окнами и фреймами).

Таблица 4.4 - Свойства, методы и события объекта ***`document`***

Свойства	Методы	События
URL	open()	Load
domain	close()	Unload
title		
lastModified	write()	Click
referrer	writeln()	DblClick
cookie		
	getSelection()	MouseDown

linkColor		MouseUp
alinkColor	getElementById()	
vlinkColor	getElementsByName()	KeyDown
	getElementsByTagName()	KeyUp
		KeyPress

Помимо перечисленных в этой таблице свойств, объект `document` имеет свойства, являющиеся коллекциями (форм, картинок, ссылок и т.п.); Таблица 3.1 содержит их описание. Кроме того, можно формировать требуемые коллекции "на лету" с помощью указанных выше методов. Так, `document.getElementsByTagName('P')` есть коллекция всех HTML-элементов (точнее, соответствующих им объектов) вида `<P>`, т.е. абзацев. Аналогично, `document.getElementsByName('important')` выдаст коллекцию (объектов) HTML-элементов любых типов, у которых был задан атрибут `NAME="important"`. Наконец, `document.getElementById('id5')` выдаст тот HTML-элемент (если их несколько, то первый), у которого был задан атрибут `ID="id5"`.

С одним методом мы уже часто работали: `document.write()` — он пишет в текущий HTML-документ. Его модификация `document.writeln()` делает то же самое, но дополнительно добавляет в конце символ новой строки; это удобно, если потом требуется читать сгенерированный HTML-документ глазами. Если запись идет в HTML-документ нового окна, открытого с помощью `window.open()`, то перед записью в него нужно открыть поток на запись с помощью метода `document.open()`, а по окончании записи закрыть поток методом `document.close()`. После выполнения последнего действия произойдет событие `Load` (и вызовется соответствующий обработчик события `onLoad`) у документа, а затем у окна.

События объекта `document` аналогичны одноименным событиям объекта `window` (только у `document` они происходят раньше), либо их смысл понятен из их названия, поэтому мы не будем детально их разбирать.

Остановимся вкратце на свойствах объекта `document`. Свойства `linkColor`, `alinkColor` и `vlinkColor` задают цвет гиперссылок — непосещенных, активных

и посещенных, соответственно. Свойство URL хранит адрес текущего документа (т.е. строковый литерал, равный window.location.href, если страница состоит из единственного документа, а не является набором фреймов). Свойство domain выдает домен (оно аналогично window.location.hostname). Свойство title выдает заголовок страницы (указанный в контейнере <TITLE>), lastModified указывает на дату и время последней модификации файла, в котором содержится данный HTML-документ (без учета времени модификации внешних файлов — стилевых, скриптов и т.п.). Свойство referrer выдает адрес страницы, с которой пользователь пришел на данную web-страницу, кликнув по гиперссылке. Наконец, свойству cookie посвящен целый раздел в лекции 8.

4.7 Фреймы (Frames)

Фреймы — это несколько видоизмененные окна. Отличаются они от обычных окон тем, что размещаются внутри них. У *фрейма* не может быть ни панели инструментов, ни меню, как в обычном окне. В качестве поля статуса *фрейм* использует поле статуса окна, в котором он размещен. Существует и ряд других отличий.

Если окно имеет фреймовую структуру (т.е. вместо контейнера <BODY> в нем присутствует контейнер <FRAMESET> со вложенными в него контейнерами <FRAME> и быть может другими контейнерами <FRAMESET>), то объект window соответствует внешнему контейнеру <FRAMESET>, а с каждым вложенным контейнером <FRAME> ассоциирован свой собственный объект класса Window.

Каждому окну или фрейму создатель страницы может дать *имя* — с помощью атрибута NAME контейнера FRAME, либо вторым аргументом метода window.open(). Используется оно в качестве значения атрибута TARGET контейнеров A и FORM, чтобы открыть ссылку или отобразить результаты работы формы в определенном окне или фрейме. Есть несколько зарезервированных имен окон: _self (имя текущего окна или фрейма, где исполняется скрипт), _blank (новое окно), _parent (окно-родитель для данного

фрейма), `_top` (самый старший предок данного фрейма, т.е. окно браузера, частью которого является данный фрейм). Иерархия фреймов, обсуждаемая ниже, как раз и задает, какие окна или фреймы являются родителями для других фреймов.

У каждого объекта класса `Window`, будь то окно или фрейм, есть также ссылка на соответствующий объект. Как мы знаем, ссылкой на объект текущего окна, в котором исполняется скрипт, является `window`; кроме того, на него же ссылается свойство `self` объекта `window` (а также свойство `window` объекта `window` — есть и такое!). Ссылку на объект окна, открываемого методом `window.open()`, выдает сам этот метод. Ссылка на объект-фрейм совпадает с его именем, заданным с помощью атрибута `NAME` контейнера `FRAME`. Наконец, у объектов-фреймов есть специальные свойства, дающие ссылки на родительский фрейм (`window.parent`) и на окно браузера, частью которого является данный фрейм (`window.top`).

Таким образом, для того, чтобы правильно обращаться к нужным фреймам, нам нужно знать лишь их *иерархию*, т.е. взаимное подчинение (какой фрейм для какого является родителем). Это мы сейчас и обсудим.

4.7.1 Иерархия и именование фреймов

Рассмотрим сначала простой пример. Разделим экран на две вертикальные колонки:

```
<HTML>
<HEAD>
<TITLE>Левый и правый</TITLE>
</HEAD>
  <FRAMESET COLS="50%,*">
     <FRAME NAME=leftframe SRC=left.htm>
     <FRAME NAME=rightframe SRC=right.htm>
  </FRAMESET>
</HTML>
```

Пример 4.6. Два фрейма

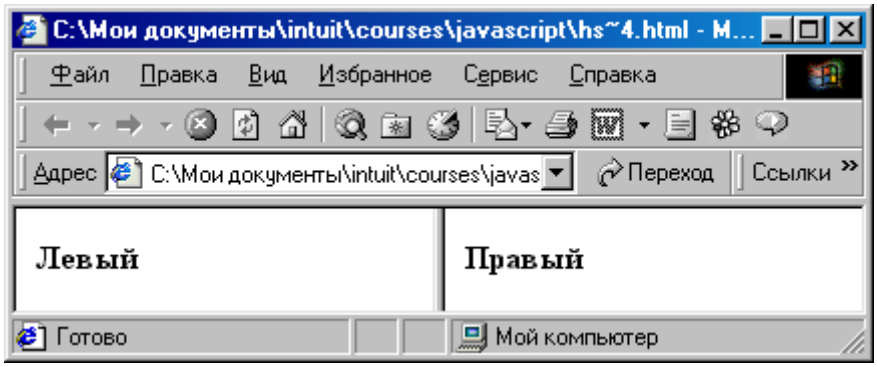


Рисунок 4.3 - Окно с двумя вертикальными фреймами

Иерархия фреймов здесь получается следующая:

- window
  - leftframe
  - rightframe

Из основного окна (из скрипта, который можно было поместить в контейнер `<HEAD>`) обратиться к левому фрейму можно с помощью `window.leftframe`, к правому — `window.rightframe`. Из каждого фрейма обратиться к основному окну можно как `window.parent` либо `window.top` (что в данном случае равносильно) или даже просто `parent` и `top` (так как приставку `window` можно опускать). Наконец, из левого фрейма обратиться к правому фрейму можно как `parent.rightframe` или `top.rightframe`.

Усложним пример: разобьем правый фрейм на два по горизонтали:

```
<HTML>
<HEAD>
<TITLE>Левый, верх и низ</TITLE>
</HEAD>
  <FRAMESET COLS="50%,*">
    <FRAME NAME=leftframe SRC=left.htm>

    <FRAMESET ROWS="50%,*">
       <FRAME NAME=topframe SRC=top.htm>
       <FRAME NAME=botframe SRC=bottom.htm>
    </FRAMESET>

  </FRAMESET>
</HTML>
```

Пример 4.7. Три фрейма

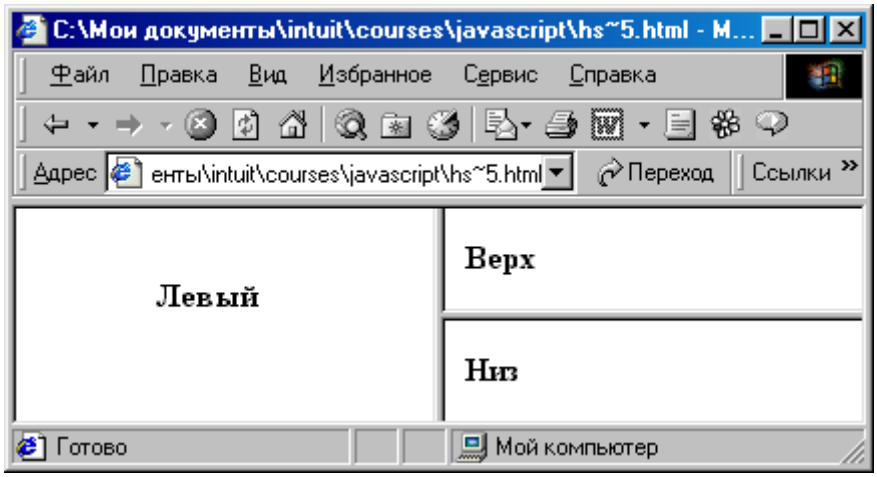


Рисунок 4.4 - Правый фрейм разбит на два по горизонтали

Фрейма с именем `rightframe` теперь не существует. Более того, все три фрейма непосредственно подчинены главному окну, т.е. иерархия выглядит следующим образом:

- window
  - leftframe
  - topframe
  - botframe

Следовательно, мы можем поместить в контейнер `<HEAD>` следующий скрипт, устанавливающий цвет фона для всех трех фреймов:

```
<SCRIPT>
window.onload=f;
function f()
{
  window.leftframe.document.bgColor='blue';
  window.topframe.document.bgColor='red';
  window.botframe.document.bgColor='green';
}
</SCRIPT>
```

Для того чтобы фрейм `rightframe` все же появился в иерархии и ему подчинялись два правых фрейма, нужно свести оба наших примера в один. Это значит, что во фрейм `rightframe` мы должны загрузить отдельный фреймовый документ.

Основной документ	Документ в правом фрейме (**`right.htm`**)
`<HTML>`	`<HTML>`

```
<HEAD>
</HEAD>
  <FRAMESET COLS="50%,*">
   <FRAME                NAME=leftframe
SRC=left.htm>
   <FRAME               NAME=rightframe
SRC=right.htm>
  </FRAMESET>
</HTML>
```

```
<HEAD>
</HEAD>
  <FRAMESET ROWS="50%,*">
     <FRAME NAME=topframe SRC=top.htm>
     <FRAME                NAME=botframe
SRC=bottom.htm>
  </FRAMESET>
</HTML>
```

В этом случае иерархия фреймов будет выглядеть иначе:

- window
  - leftframe
  - rightframe
    - topframe
    - botframe

Теперь чтобы из главного окна обратиться ко всем трем фреймам и установить в них те же цвета фона, следует писать:

```
window.leftframe.document.bgColor='blue';
window.rightframe.topframe.document.bgColor='red';
window.rightframe.botframe.document.bgColor='green';
```

Таким образом, визуально на Web-странице мы получили тот же результат, что и с тремя фреймами, подчиненными одному старшему окну (см. пример 4.7). Однако этот вариант более гибкий: он позволяет работать независимо с фреймом `rightframe` в отдельном файле.

4.7.2 Коллекция фреймов

Выше мы обращались к фрейму по его имени. Однако, если имя не известно (или не задано), либо если нужно обратиться ко всем дочерним фреймам по очереди, то более удобным будет обращение через *коллекцию* фреймов `frames[]`, которая является свойством объекта `window`.

В качестве иллюстрации предположим, что в примере из двух фреймов (пример 4.6) правый фрейм содержит несколько изображений, и нам требуется поменять адрес (значение атрибута `SRC`) третьего изображения с

помощью скрипта, находящегося в левом фрейме. Правый фрейм — второй, значит, его номер 1; третье изображение имеет номер 2. Поэтому, это можно сделать следующими способами:

```
top.frames[1].document.images[2].src = 'pic.gif';
top.frames['rightframe'].document.images[2].src = 'pic.gif';
top.frames.rightframe.document.images[2].src = 'pic.gif';
top.rightframe.document.images[2].src = 'pic.gif';
```

4.7.3 Передача данных во фрейм

Обычной задачей при разработке типового Web-узла является загрузка результатов исполнения CGI-скрипта во *фрейм*, отличный от фрейма, в котором вводятся данные для этого скрипта. Если путь загрузки результатов фиксированный, то можно просто использовать атрибут `TARGET` формы. Сложнее, если результат работы должен быть загружен в разные фреймы (например, в зависимости от выбранной кнопки).

Применим полученные нами знания для решения этой задачи. Сначала заготовим следующие файлы. Основной файл, например, `index.htm`, содержит левый фрейм, в котором будет находиться форма, и правый фрейм, разбитый на два подфрейма (верхний и нижний). Файл `left.htm` содержит форму, в которой пользователю предоставляется возможность выбрать верхний или нижний фрейм и нажать кнопку "Загрузить". Файл `right.htm` содержит простой текст; он будет загружаться в верхний или нижний фрейм, в зависимости от действий пользователя.

Основной файл с тремя фреймами	Файл с формой **left.htm** в левом фрейме	Файл **right.htm**

`<HTML>`
`<HEAD>`
`<TITLE>Три фрейма</TITLE>`
`</HEAD>`
`<FRAMESET COLS="50%,*">`
`<FRAME NAME=leftframe`
`SRC=left.htm>`
	`<HTML>`
	`<HEAD>`
	`<SCRIPT`
	`SRC="loadframe.js"></SCRIPT>`
	`</HEAD>`
	`<BODY>`
	`<FORM METHOD=post ACTION=right.htm`
	`NAME=f onSubmit="return load();">`
		`<HTML>`
		`<BODY>`
		
		`Этот документ мы загружаем при выборе фрейма`

```
    <FRAMESET
ROWS="50%,*">
     <FRAME   NAME=topframe
SRC="">
     <FRAME   NAME=botframe
SRC="">
    </FRAMESET>

  </FRAMESET>
</HTML>
```

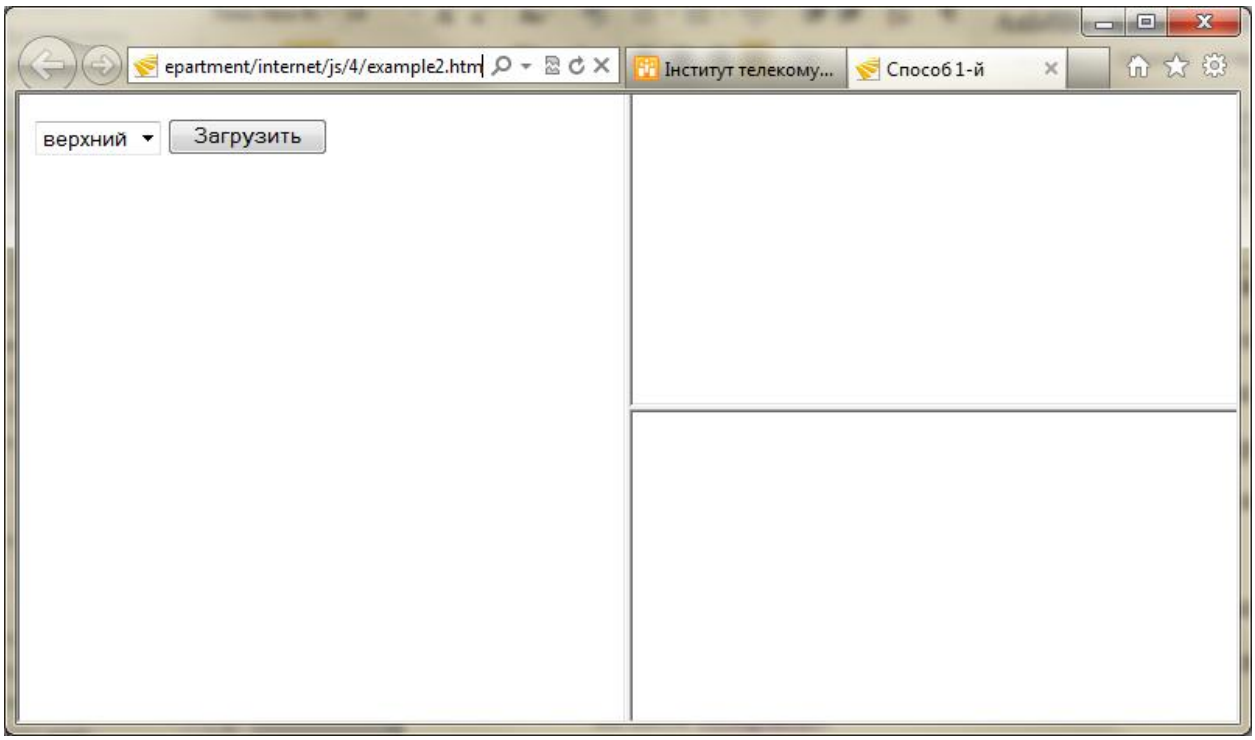
```
  <SELECT NAME=s>
       <OPTION>верхний</OPTION>
       <OPTION>нижний</OPTION>
  </SELECT>
 <INPUT                  TYPE=submit
VALUE="Загрузить">
 </FORM>
</BODY>
</HTML>
```

```
из списка


</BODY>
</HTML>
```

Для того, чтобы пример заработал, остается в файле `loadframe.js` описать функцию `load()`. Функция должна делать так, чтобы в зависимости от выбора пользователем значения селектора "верхний" или "нижний" файл `right.htm` загружался бы либо в правый верхний, либо в правый нижний фрейм. С этой целью в файле `left.htm` у формы не был указан целевой фрейм (атрибут `TARGET`).

Нашу задачу динамического выбора фрейма можно решать по-разному. Более изящный способ — переназначать "на лету" свойство `target`, с него мы и начнем (открыть).




```
function load()
{
if(document.f.s.selectedIndex==0)
```

```
  {
   document.f.target = "topframe";
   top.frames[2].document.open();
   top.frames[2].document.close();
  }
else
  {
   document.f.target = "botframe";
   top.frames[1].document.open();
   top.frames[1].document.close();
  }
return true;
}
```

Пример 4.8. Файл loadframe.js: переназначение target на лету

Функция `load()` всегда возвращает `true`, а поскольку она вызывается из обработчика события `onSubmit`, это означает, что всегда будет происходить отправка формы (событие `Submit`), т.е. загрузка страницы `right.htm`, указанной в атрибуте `ACTION` данной формы. Обратите внимание также на следующие строки в функции `load()`:

```
   top.frames[1].document.open();
   top.frames[1].document.close();
```

Смысл их таков: когда пользователь выбирает значение `верхний` или `нижний` в форме, то файл `right.htm` загружается в соответствующий фрейм, а оставшийся фрейм открывается на запись (методом `...document.open()`, при этом всё его содержимое очищается) и закрывается (методом `...document.close()`), тем самым фрейм остаётся пустым (без текста).

Теперь рассмотрим второй подход — открытие окна с именем, совпадающим с именем фрейма `topframe` или `botframe`. Его идея состоит в том, что при попытке открыть окно с именем существующего окна новое окно не открывается, а используется уже открытое. Фрейм — это тоже окно, поэтому на него данное правило распространяется. Функция, реализующая такое поведение, приведена ниже (открыть):

```
function load()
{
  if(document.f.s.selectedIndex==0)
  {
```

```
    window.open("right.htm","topframe");
    top.frames[2].document.open();
    top.frames[2].document.close();
  }
  else
  {
    window.open("right.htm","botframe");
    top.frames[1].document.open();
    top.frames[1].document.close();
  }
return false;
}
```

Пример 4.9. Файл loadframe.js: использование window.open()

В этом подходе функция `load()` всегда возвращает `false`. Это необходимо, чтобы отменить отправку данных формы: ведь после того, как мы вызвали `window.open()`, в отправке данных формы, т.е. загрузке файла `right.htm`, уже нет надобности.

5 ЛЕКЦИЯ: ПРОГРАММИРУЕМ ФОРМЫ

Рассматривается самая старая часть спецификации JavaScript — программирование HTML-форм. Разбираются различные методы обработки событий, перехват отправки данных на сервер и способы организации обмена данными при помощи форм и JavaScript-кода.

5.1 Контейнер FORM

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

```
<FORM NAME=fname METHOD=get>
<INPUT NAME=iname SIZE=30 MAXLENGTH=30>
</FORM>
```

Поэтому в программах на **JavaScript** к ним обращаются по имени:

```
document.fname.iname.value="Текст";
```

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

```
document.forms[0].elements[0].value="Текст";
document.forms['fname'].elements['iname'].value="Текст";
```

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

Свойства, методы и события объекта `Form`		
Свойства	**Методы**	**События**

`length`
`action`
`method`
`target`
`encoding`
`elements[]`
	`reset()`
	`submit()`
		`Reset`
		`Submit`

5.2 Свойства объекта Form

5.2.1 Свойство action

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

```
<FORM METHOD=post ACTION="javascript: alert('Работает!');">
<INPUT TYPE=submit VALUE="Продемонстрировать JavaScript в ACTION">
</FORM>
```

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

```
alert('Строка');?
```

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

5.2.2 Свойство method

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

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

```
<FORM NAME=f ACTION="javascript: alert('Работает!');">
<SCRIPT>
document.write('По умолчанию установлен метод: '+document.f.method+'.<BR>');
</SCRIPT>
<INPUT TYPE=button onClick="document.f.method='post'" VALUE="Сменить метод на
POST">
<INPUT TYPE=button onClick="document.f.method='get';" VALUE="Сменить метод на
GET"><BR>
<INPUT TYPE=submit VALUE="JavaScript в ACTION">
</FORM>
```

Пример 5.1. Изменение метода формы (GET и POST) скриптом

По умолчанию установлен метод GET.

В данном примере стоит обратить внимание на два момента:

1. Прежде чем открывать окно предупреждения, следует нажать кнопку "Метод **POST**". Если этого не сделать, то появится сообщение об ошибке **JavaScript**. Здесь все выглядит достаточно логично. Формирование **URL** происходит при генерации события **submit**, а вызов скрипта — после того, как событие сгенерировано. Поэтому вставить переопределение метода в обработчик события нельзя, так как к этому моменту будет уже сгенерирован **URL**, который, в свою очередь, будет **JavaScript**-программой с символом "?" на конце. Переопределение метода должно быть выполнено раньше, чем произойдет событие `Submit`.
2. В тело документа через контейнер `SCRIPT` встроен **JavaScript**-код, который сообщает *метод доступа*, установленный в форме по умолчанию. Этот контейнер расположен сразу за контейнером `FORM`. Ставить его перед контейнером `FORM` нельзя, так как в момент получения интерпретатором управления объект `FORM` не будет создан, и, следовательно, работать с его свойствами не представляется возможным.

Никаких других особенностей свойство `method` не имеет. В данном свойстве можно указать и другие методы доступа, отличные от `GET` и `POST`, но это требует дополнительной настройки сервера.

5.2.3 Свойство target

Свойство `target` определяет имя окна, в которое следует загружать результат обращения к **CGI**-скрипту. При этом всегда есть альтернативы: можно использовать значение этого свойства внутри **JavaScript**-программ для указания окна или фрейма, куда требуется загружать результат работы **CGI**-скрипта, а можно получить идентификатор окна или задействовать встроенный массив `frames[0]` и свойства окна `opener`, `top` и `parent`. Кроме того, для загрузки внешнего файла в некоторое окно или фрейм можно также применить метод `window.open()`. Все эти варианты будут продемонстрированы в разделе "Передача данных во фрейм".

5.2.4 Свойство encoding

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

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

5.2.5 Коллекция elements[]

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

```
<FORM NAME=f>
<INPUT NAME=e SIZE=40>
<BR><INPUT TYPE=button VALUE="Ввести текст по имени элемента"
  onClick="document.f.e.value='Текст введен по имени элемента';">
<BR><INPUT TYPE=button VALUE="Ввести текст по индексу элемента"
  onClick="document.f.elements[0].value='Текст введен по индексу элемента';">
<BR><INPUT TYPE=reset VALUE="Очистить">
</FORM>
```

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

5.3 Методы объекта Form

5.3.1 Метод submit()

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

```
<FORM NAME=f ACTION="http://its.kpi.ua/rating_students/">
Ваше имя пользователя на its:<INPUT NAME=query>
</FORM>
<A HREF="javascript:document.f.submit();">Посмотреть рейтинг</A>
```

Как видите, кнопки отправки (submit) у формы нет, но нажав на ссылку, мы выполняем отправку данных на сервер. Обычно при такой "скрытой" отправке данных на сервер браузеры, в целях безопасности, запрашивают подтверждение, действительно ли пользователь желает отправить данные. Отправка данных путем вызова метода submit() имеет отличия от нажатия пользователем кнопки INPUT типа TYPE=submit; их мы рассмотрим в конце лекции.

5.3.2 Метод reset()

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

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

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

5.4 События объекта Form

5.4.1 Событие Submit

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

Фукцию обработки этого события можно переопределить и даже вовсе отменить. Для этой цели введен атрибут onSubmit="код_программы" у контейнера <FORM>. В нем можно указать действия (**JavaScript**-код), какие должны выполняться при возникновении этого события. Порядок выполнения этих действий и действий браузера, а также использование оператора return false для отмены последних, полностью аналогичны тем, что описаны ниже для onReset. Пример:

```
<SCRIPT>
function TestBeforeSend()
{
 if(document.f.query.value=='')
 {
   alert('Пустую строку не принимаем!');
   return false;
 }
 else return true;
}
</SCRIPT>

<FORM NAME=f METHOD=post onSubmit="return TestBeforeSend();"
      ACTION="http://its.kpi.ua/rating_students/">
Ваше имя пользователя на ITS:<INPUT NAME=query>
<INPUT TYPE=submit VALUE="Посмотреть рейтинг">
</FORM>
```

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

5.4.2 Событие Reset

Событие Reset возникает (и соответствующий обработчик события onReset вызывается) при нажатии пользователем на кнопку типа reset или при выполнении метода reset(). Действие по умолчанию, которое выполняет браузер при возникновении этого события — восстановление значений по умолчанию в полях формы. Однако функцию обработки этого события можно переопределить и даже вовсе отменить. Для этой цели введен атрибут onReset="код_программы" у контейнера <FORM>. В нем можно указать действия (**JavaScript**-код), какие должны выполняться при

возникновении этого события. Браузер сначала выполняет эти действия, а затем — свое действие по умолчанию. Но если последним оператором в обработчике onReset будет return false, то действие браузера по умолчанию выполняться не будет. Этот прием называется *перехватом события*. Пример:

```
<FORM onReset="javascript: alert('Не дадим восстановить!');return false;">
<INPUT VALUE="Измените этот текст" SIZE=30>
<INPUT TYPE=reset VALUE="Восстановить">
</FORM>
```

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

5.5 Поля формы и их объекты

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

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

При программировании форм часто требуется писать обработчики событий для форм или их элементов, при этом нужно ссылаться на свойства данного элемента, других элементов и формы в целом. Стандартная схема именования по именам либо по индексам обсуждалась выше:

```
document.форма.элемент.свойство      // точечная нотация
document.форма.элемент["свойство"]   // скобочная нотация
document.forms["имя_формы"].elements["имя_элемента"].свойство
document.forms[индекс_формы].elements[индекс_элемента].свойство
```

Однако получающиеся выражения — довольно громоздкие. Поэтому было введено следующее **соглашение**: в обработчике события формы или элемента формы имя **текущего** элемента можно опускать (вместе со всей предшествующей "приставкой"). Кроме того, ссылаться на сам текущий элемент можно с помощью ключевого слова this. Последнее может потребоваться, например, когда нужно передать в функцию не какое-то *свойство* объекта, а сам объект.

Например, предположим, что у нас есть форма:

```
<FORM NAME=f>
<INPUT TYPE=text   NAME=e value="Текст"  onFocus="">
<INPUT TYPE=button NAME=b value="Кнопка" onClick="">
</FORM>
```

Тогда вместо полной записи:

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

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

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

Более того, в этом контексте эквивалентны следующие записи:

```
                    value  // короче не бывает!
               this.value  // здесь this ссылается на элемент "e"
             form.e.value  // form есть свойство объекта "e" (равное "f")
        this.form.e.value  // комбинируем оба способа
       document.f.e.value  // почти полная запись
window.document.f.e.value  // это самая полная запись
document.f.e.form.e.value  // можно итерировать "form.e."
```

Например, здесь в 3-й строчке form есть свойство (текущего!) элемента document.f.e — напомним, что это свойство ссылается на объект document.f. Аналогично, в обработчик onClick элемента b мы можем поместить скрипт form.e.value=50 (краткое обращение к свойству другого элемента формы: document.f.e.value) или alert(form.method) (краткое обращение к свойству самой формы document.f.method) или даже TestForBugs(this) (в пользовательскую функцию TestForBugs() будет передан (по ссылке) объект document.f.b).

Как видим, это соглашение не только дает некоторую экономию кода, но также позволяет ссылаться на текущий элемент или на форму, не зная его имени или номера. Это предоставляет дополнительную гибкость при программировании форм; например, можно переименовать форму, ничего не меняя во всех скриптах. Далее мы рассмотрим объекты **JavaScript**, соответствующие полям различных типов в **HTML**-формах. При этом мы будем пользоваться данным соглашением. Поскольку свойства name, type и form есть у объектов всех элементов формы, то мы не будем их указывать особо.

5.5.1 Текстовое поле ввода (объект Text)

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

```
<A HREF="http://site.com/">ссылка 1</A>
<FORM>Число гипертекстовых ссылок к данному моменту:
<SCRIPT>
document.write('<INPUT NAME=t VALUE='+document.links.length+'>');
</SCRIPT>
<BR><INPUT TYPE=button
VALUE="Число ссылок по окончании загрузки страницы"
onClick="form.t.value=document.links.length;">
<BR><INPUT TYPE=reset>
</FORM>
<A HREF="http://rite.com/">ссылка 2</A>
```

Пример 5.2.

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

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

Свойства, методы и события объекта Text			
Свойства	**Методы**	**Обработчики событий**	
defaultValue value size maxLength disabled readOnly	focus() blur() select()	onChange onSelect onFocus onBlur onClick onDblClick	onMouseOver onMouseOut onMouseDown onMouseUp onKeyPress onKeyDown onKeyUp

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

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

Смысл *обработчиков событий* вполне понятен из их названий: обработчик onChange вызывается, когда пользователь (но не скрипт) изменил значение в поле ввода (и кликнул вне поля ввода); onSelect — когда пользователь начинает выделять текст, расположенный в поле; onFocus и onBlur — когда поле получает и теряет фокус, соответственно; onClick и onDblClick — когда пользователь совершил одинарный или двойной щелчок мышью на поле, соответственно. Вторая колонка событий — стандартна для большинства элементов **HTML**-страницы. Нужно лишь иметь в виду, что обработчики событий onMouseDown, onMouseUp, onKeyPress, onKeyDown, onKeyUp срабатывают у того элемента формы, который в данный момент находится в фокусе.

5.5.2 Списки вариантов (объекты Select и Option)

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

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

Объект Select		
Свойства	**Методы**	**Обработчики событий**
options[] size length multiple selectedIndex	focus() blur() add() remove()	onBlur onChange onFocus

Объект Option		
Свойства	**Методы**	**События**
defaultSelected selected index text value	нет	нет

Мы не будем описывать все свойства, методы и события этих двух объектов. Остановимся только на типичных способах применения их комбинаций.

5.5.3 Создание объектов Option

Объект класса `Option` интересен тем, что в отличие от многих других встроенных в **DOM** объектов **JavaScript**, имеет конструктор. Это означает, что программист может сам создать объект класса `Option`:

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

где аргументы соответствуют свойствам обычных объектов класса `Option`:

- `text` — строка текста, которая размещается в контейнере `<OPTION>` (например: `<OPTION>`текст`</OPTION>`);
- `value` — значение, которое передается серверу при выборе альтернативы, связанной с объектом `Option`;
- `defaultSelected` — выбрана ли эта альтернатива по умолчанию (`true`/`false`);
- `selected` — альтернатива была выбрана пользователем (`true`/`false`).

На первый взгляд не очень понятно, для чего может понадобиться программисту такой объект, ведь создать объект класса `Select` нельзя и, следовательно, нельзя приписать ему новый объект `Option`. Все объясняется, когда речь заходит об изменении списка альтернатив у имеющегося в документе объекта `Select`. Делать это можно, при этом изменение списка альтернатив `Select` не приводит к переформатированию документа. Изменение списка альтернатив позволяет решить проблему создания вложенных меню, которых нет в **HTML**-формах, путем программирования обычных меню (`options[]`).

При программировании альтернатив следует обратить внимание на то, что у объектов класса `Option` нет свойства `name`, в виду того, что у контейнера `<OPTION>` нет атрибута `NAME`. Таким образом, к встроенным в документ объектам класса `Option` можно обращаться только как к элементам коллекции `options[]`.

5.5.4 Коллекция options[]

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

Программист имеет возможность не только создавать новые объекты Option, но и удалять уже созданные браузером объекты:

```
<FORM>
<SELECT NAME=s>
<OPTION>Первый вариант</OPTION>
<OPTION>Второй вариант</OPTION>
<OPTION>Третий вариант</OPTION>
</SELECT>
<INPUT TYPE=button VALUE="Удалить последний вариант"
       onClick="form.s.options[form.s.length-1]=null;">
<INPUT TYPE=reset VALUE="Сбросить">
</FORM>
```

Пример 5.3. Удаление вариантов из SELECT

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

Теперь, используя конструктор Option, сделаем процесс обратимым:

```
<SCRIPT>
function RestoreOptions()
{
document.f.s.options[0] = new Option('Вариант один','',true,true);
document.f.s.options[1] = new Option('Вариант два');
document.f.s.options[2] = new Option('Вариант три');
return false;
}
</SCRIPT>
<FORM NAME=f onReset="RestoreOptions();">
<SELECT NAME=s>
<OPTION>Первый вариант</OPTION>
<OPTION>Второй вариант</OPTION>
<OPTION>Третий вариант</OPTION>
</SELECT>
<INPUT TYPE=button VALUE="Удалить последний вариант"
      onClick="form.s.options[form.s.length-1]=null;">
<INPUT TYPE=reset VALUE=Reset>
</FORM>
```

Пример 5.4. Удаление и добавление вариантов из SELECT

Восстановление альтернатив мы поместили в обработчик события onReset контейнера FORM. Создаваемые объекты класса Option мы подчиняем объекту document.f.s класса

Select. При этом первая альтернатива должна быть выбранной по умолчанию (аргументу defaultSelected задано значение true), чтобы смоделировать поведение при начальной загрузке страницы.

5.5.5 Свойства text и value объекта Option

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

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

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

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

5.5.6 Свойства selected и selectedIndex

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

```
<FORM> Вариант:
<SELECT onChange="form.e.value=selectedIndex;">
<OPTION>Один</OPTION>
<OPTION>Два</OPTION>
</SELECT>
Выбрали индекс: <INPUT NAME=e>
</FORM>
```

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

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

5.5.7 Обработчик события onChange объекта Select

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

```
<FORM>
Фрукты: <SELECT MULTIPLE
onChange="form.e.value='';
          for(i=0; i<length; i++)
          if(options[i].selected)
          form.e.value += options[i].text+', ';">

<OPTION>яблоко</OPTION>
<OPTION>банан</OPTION>
<OPTION>киви</OPTION>
<OPTION>персик</OPTION>
</SELECT><BR>
Выбраны позиции: <INPUT READONLY SIZE=70 NAME=e>
</FORM>
```

Пример 5.5. Обработчик onChange при выборе множественных вариантов

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

5.6 Кнопки

В **HTML**-формах используется четыре вида кнопок:

```
<FORM>
<INPUT TYPE=button VALUE="Кнопка типа button">
<INPUT TYPE=submit VALUE="Кнопка отправки">
<INPUT TYPE=reset  VALUE="Кнопка сброса">
<INPUT TYPE=image SRC=a.gif> <!-- графическая кнопка -->
</FORM>
```

В атрибуте кнопки можно задать обработчик события onClick, а в атрибуте формы — обработчики событий onSubmit и onReset. Кроме того, кнопкам и форме соответствуют объекты **DOM**. Объект, отвечающий кнопке, имеет метод click(). Объект, отвечающий форме, имеет методы submit() и reset(). С точки зрения программирования важен вопрос о взаимодействии этих методов друг с другом и с соответствующими обработчиками событий.

В каком случае при вызове метода (из любого места **JavaScript**-программы) будет автоматически вызван и соответствующий обработчик события, заданный пользователем в атрибуте кнопки или формы? Ответ здесь следующий:

- при вызове метода click() кнопки вызывается и обработчик события onClick этой кнопки;
- при вызове метода submit() формы **не вызывается** обработчик события onSubmit формы;
- при вызове метода reset() формы вызывается и обработчик события onReset формы.

Ниже мы на примерах рассмотрим, что это означает на практике. Таким образом, при программном вызове метода submit() нужно позаботиться о дополнительном вызове обработчика события onSubmit, чтобы, например, данные не были отправлены на сервер без предварительной проверки. Как это сделать — мы расскажем ниже. Особое внимание мы уделим также возможности перехвата и генерирования события отправки данных на сервер.

5.6.1 Кнопка button

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

```
<FORM NAME=f>
<INPUT TYPE=button NAME=b VALUE="Кнопка" onClick="alert('5+7='+(5+7))">
</FORM>
<A HREF="javascript:document.f.b.click();void(0);">Вызвать метод click()</A>
```

Вызов метода click() у кнопки равносилен нажатию кнопки, что и демонстрирует приведенный пример. Как мы увидим ниже, это же справедливо для любых типов кнопок.

5.6.2 Кнопка submit

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

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

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

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

Теперь усложним наш пример: добавим обработчики событий onClick (у кнопки отправки) и onSubmit (у формы), и посмотрим на поведение браузера при нажатии кнопки отправки:

```
<FORM NAME=f ACTION="receive.htm"
      onSubmit="return confirm('Вы хотите отправить данные?')">
<INPUT onClick="alert('Вызван обработчик onClick у кнопки отправки')"
       TYPE=submit VALUE="Кнопка отправки" NAME=s>
</FORM>
```

Пример 5.6. Обработчики onClick у кнопки отправки и onSubmit у формы

Убедитесь, что нажатие кнопки отправки приводит к следующей последовательности действий браузера:

1. вызов обработчика события onClick у данной кнопки;
2. вызов обработчика события onSubmit у формы;
3. отправка данных формы на сервер.

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

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

```
<FORM NAME=f ACTION="receive.htm"
      onSubmit="return confirm('Вы хотите отправить данные?')">
<INPUT onClick="alert('Вызван обработчик onClick у кнопки отправки')"
       TYPE=submit VALUE="Кнопка отправки" NAME=s></FORM>

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

Пример 5.7. Вызов метода click() у кнопки отправки

5.6.3 Метод submit() формы

Вызов метода submit() формы **не равносилен** нажатию кнопки отправки. При вызове этого метода будет выполнено только третье из вышеперечисленных трех действий — отправка данных на сервер. То, что он не должен порождать вызов обработчика onClick кнопки отправки, вполне понятно — ведь мы пытаемся отправить данные в обход кнопки отправки (которой, кстати, может и не быть вовсе). Но и обработчик события onSubmit у формы тоже **не вызывается** — это является для многих неожиданным. Не будем судить, насколько это логично (и почему это поведение отличается от поведения метода reset(), см. ниже), а просто проиллюстрируем этот эффект, введя в предыдущий пример ссылку, вызывающую метод submit():

```
<FORM NAME=f ACTION="receive.htm"
      onSubmit="return confirm('Вы хотите отправить данные?')">
<INPUT onClick="alert('Вызван обработчик onClick у кнопки отправки')"
       TYPE=submit VALUE="Кнопка отправки" NAME=s></FORM>

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

Пример 5.8. Метод submit() не вызывает обработчика onSubmit

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

```
<FORM NAME=f ACTION="receive.htm"
      onSubmit="return confirm('Вы хотите отправить данные?')">
<INPUT onClick="alert('Вызван обработчик onClick у кнопки отправки')"
       TYPE=submit VALUE="Кнопка отправки" NAME=s></FORM>

<A HREF="javascript:
    if(document.f.onsubmit() != false)
       document.f.submit();  void(0);"
 >Вызвать <B>submit()</B> с преварительной проверкой onSubmit</A>
```

Пример 5.9. Принудительный вызов onSubmit перед submit()

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

```
Введите текст и нажмите Enter:<BR>
<FORM ACTION="receive.htm"
   onSubmit="return confirm('Вы хотите отправить данные?')">
<INPUT TYPE=text VALUE="Текст вводить здесь:" SIZE=50>
</FORM>
```

Пример 5.10. Отправка данных формы нажатием клавиши Enter

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

5.6.4 Кнопка reset

Кнопка сброса (`reset`) позволяет вернуть все поля формы в первоначальное состояние, которое они имели при загрузке страницы. Нажатие кнопки сброса приводит к следующей последовательности действий браузера:

1. вызов обработчика события `onClick` у данной кнопки;
2. вызов обработчика события `onReset` у формы;
3. восстановление значений по умолчанию во всех полях формы.

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

```
<FORM NAME=f
   onReset="return confirm('Вы хотите очистить форму?')">
<INPUT TYPE=text VALUE="Измените этот текст">
<INPUT TYPE=reset VALUE="Кнопка сброса" NAME=s
  onClick="alert('Вызван обработчик onClick у кнопки сброса')">
</FORM>
<A HREF="javascript: document.f.s.click();void(0);"
 >Вызвать метод <B>click()</B> кнопки сброса</A>
```

Пример 5.11. Вызов метода click() у кнопки сброса

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

```
Измените текст, а затем нажмите Esc (либо ссылку).<BR>
<FORM NAME=f
   onReset="return confirm('Вы хотите очистить форму?')">
<INPUT TYPE=text VALUE="Измените этот текст">
</FORM>
<A HREF="javascript: document.f.reset();void(0);"
 >Вызвать метод <B>reset()</B> формы</A>
```

Пример 5.12. Сброс формы нажатием клавиши Esc

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

5.6.5 Графическая кнопка

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

```
<FORM ACTION="receive.htm">
<INPUT TYPE=image SRC="pic.gif">
</FORM>
```

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

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

Как же обратиться к такой кнопке**?** Это можно сделать, например, задав атрибут `ID`:

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

и затем в программе написав: `var knopka = document.getElementById('d1')`. После этого мы можем обращаться к свойствам этой кнопки, например `knopka.src`, а также к методу `knopka.click()`. Следующий пример показывает, что вызов метода `click()` графической кнопки "почти" равносилен нажатию этой кнопки, т.е. последовательно вызывает обработчики `onClick` кнопки, `onSubmit` формы и передает данные на сервер **(**но что при этом передается в качестве координат курсора мыши**?):**

```
<FORM ACTION="receive.htm"
      onSubmit="return confirm('Вы хотите отправить данные?')">
<INPUT onClick="alert('Вызван обработчик onClick у графической кнопки')"
      TYPE="image" SRC="pic.gif" id="d1">
</FORM>

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

Пример 5.13. Вызов метода click() у графической кнопки

Поскольку графические кнопки используются довольно редко, на этом мы остановимся в их обсуждении.

6 ЛЕКЦИЯ: ПРОГРАММИРУЕМ ГИПЕРТЕКСТОВЫЕ ПЕРЕХОДЫ

Рассматриваются вопросы работы с коллекцией гипертекстовых ссылок и программирования гипертекстовых переходов в зависимости от условий просмотра HTML-страниц и действий пользователя.

Для начала нам нужно разделить несколько понятий: применимость URL в атрибутах HTML-контейнеров; коллекция гипертекстовых ссылок; объекты класса `URL`.

Адреса URL могут использоваться в атрибутах многих **HTML-**контейнеров**,** например**:**

- ссылки **(URL** в атрибуте `HREF` контейнера `<A>`**);**
- активные области **(URL** в атрибуте `HREF` контейнера `<AREA>`**);**
- картинки **(URL** в атрибуте `SRC` контейнера `<IMG>`**);**
- формы **(URL** в атрибуте `ACTION` контейнера `<FORM>`**);**
- внешние скрипты **(URL** в атрибуте `SRC` контейнера `<SCRIPT>`**);**
- связанные документы **(URL** в атрибуте `HREF` контейнера `<LINK>`**).**

Гипертекстовая ссылка в **HTML-**документе **—** это область **HTML-**страницы**,** по которой можно **"**кликнуть**" (**или выбрать ее иным способом**),** чтобы перейти к просмотру другого **HTML-**документа**.** Из всех перечисленных выше вариантов применения **URL** гипертекстовыми ссылками являются лишь первые два**.** В объектной модели документа **(DOM)** они собраны в единую *коллекцию* гипертекстовых ссылок `document.links[]`**.** Нумерация в ней начинается с нуля **(**как обычно**),** в порядке появления ссылок в документе**.**

Обратите внимание**,** что в принципе могут существовать ссылки**,** по которым невозможно **"**кликнуть**",** т**.**к**.** они занимают нулевую площадь **web-**страницы**;** например**,** контейнер `<A>` с пустым содержимым**,** т**.**е**.** `<A HREF="http://ya.ru/"></A>`**,** или контейнер `<AREA>`**,** ограничивающий фигуру нулевой площади**.** Тем не менее все они считаются гипертекстовыми ссылками и содержатся в коллекции `document.links[]`**.** С другой стороны**,** в коллекцию `document.links[]` **не** попадают *якоря*, то есть контейнеры `<A>`**,** не имеющие атрибута `HREF` **(**якоря нужны**,** чтобы задать место**,** на которое можно сослаться из другой гиперссылки**).** Все контейнеры `<A>` **(**как якоря**,** так и гиперссылки**)** собраны в коллекции `document.anchors[]`**;** в этой лекции**,** однако**,** она нас не будет интересовать**.**

В объектной модели документа **DOM** объекты ***класса URL*** имеют такие свойства**,** как `href`**,** `protocol`**,** `hostname` и т**.**д**. (**полный перечень см**.** ниже**).** В класс `URL` входят объекты коллекции `document.links[]`**,** а также объект `window.location`**,** рассматривавшийся в лекции **4****.** Конечно**,** помимо общих свойств**,** перечисленных ниже**,** эти объекты могут иметь свои специфичные только для них свойства**,** методы и события**.** Например**,** у объекта `window.location` есть метод `reload()`**,** тогда как у ссылок его нет**,** но у них есть обработчик события `onClick`**.** Мы начнем с рассмотрения объектов класса `URL`**.**

6.1 Объекты URL

Объект класса URL обладает свойствами, которые определены схемой **URL.** В качестве примера рассмотрим ссылку:

```
http://www.site.ru:80/dir/page.cgi?product=phone&id=3#mark
```

Тогда ее свойства примут следующие значения **(**обратите внимание, что значение поля search начинается со знака "?", а значение hash — со знака "#")

Свойства объекта URL	
Свойство	**Значение**
href	**http://www.site.ru:80/dir/page.cgi?product=phone&id=3#mark**
protocol	**http:**
hostname	**www.site.ru**
port	**80**
host	**www.site.ru:80**
pathname	**dir/page.cgi**
search	**?product=phone&id=3**
hash	**#mark**

Как Вы помните из прошлых лекций, к свойствам можно обращаться, используя точечную нотацию **(**document.links[0].host**)** или скобочную нотацию **(**document.links[0]["host"]**).** Свойства объекта класса URL дают программисту возможность менять только часть **URL-**адреса. Наиболее интересно это выглядит в объекте window.location, когда при изменении его свойства href происходит загрузка нового документа.

Свойство href является свойством **по умолчанию.** Это значит, что вместо window.location.href="..." можно писать window.location="...", а опуская window **(**который является объектом по умолчанию**),** можно писать location.href="..." и даже location="..." — эффект будет тот же: загрузится страница с новым адресом. С этим, однако, стоит быть осторожнее, чтобы данный оператор присваивания не находился в контексте, где объектом по умолчанию может быть какой-либо другой объект, например, внутри оператора with.

Обратите внимание, что свойства объекта URL взаимозависимы, точнее, свойство href зависит от остальных свойств, а остальные свойства зависят от href. Так, если присвоить новый **URL** свойству href объекта, то автоматически изменятся и все остальные свойства, разобрав данный **URL** на составные части. И наоборот, если,

например, изменить значение свойства protocol с http: на ftp:, то изменится и значение свойства href.

6.2 Коллекция ссылок links[]

К встроенным *гипертекстовым ссылкам* относятся собственно ссылки (<A HREF=...>...</A>) и ссылки "чувствительных" графических картинок. Все вместе они составляют *коллекцию* (встроенный массив) *гипертекстовых ссылок* документа document.links[].

К сожалению, обратиться по имени к гипертекстовой ссылке (т.е. как document.имя_ссылки) нельзя, даже несмотря на то, что у ссылки может быть задан атрибут NAME. Говоря точнее, такое обращение не рекомендуется в силу различий между браузерами. Поэтому обращаться к ним можно только через коллекцию ссылок по индексу: document.links[3] — это **4**-я ссылка в документе. Стандарт также предусматривает обращение к ссылкам через коллекцию по имени: document.links["имя_ссылки"], однако это работает не во всех браузерах (в **Mozilla Firefox** работает, в **IE7** нет). Поэтому в дальнейшем, в целях совместимости, мы будем обращаться к ссылкам через коллекцию по их индексу.

В качестве примера распечатаем гипертекстовые ссылки некоторого документа:

```
for(i=0;i<document.links.length;i++)
document.write(document.links[i].href+"<BR>");
```

В результате можем получить список, например, такой:

```
http://its.kpi.ua/help/index.html
http://its.kpi.ua/help/terms.html
http://its.kpi.ua/help/shop.html
```

Вставим в документ контейнер MAP с двумя ссылками, привязанными к областям (даже) нулевого размера:

```
<MAP NAME=test>
 <AREA SHAPE=rect COORDS="0,0,0,0" HREF="javascript:alert('Область 1')">
 <AREA SHAPE=rect COORDS="0,0,0,0" HREF="javascript:alert('Область 2')">
</MAP>
```

И снова распечатаем массив ссылок — получим уже:

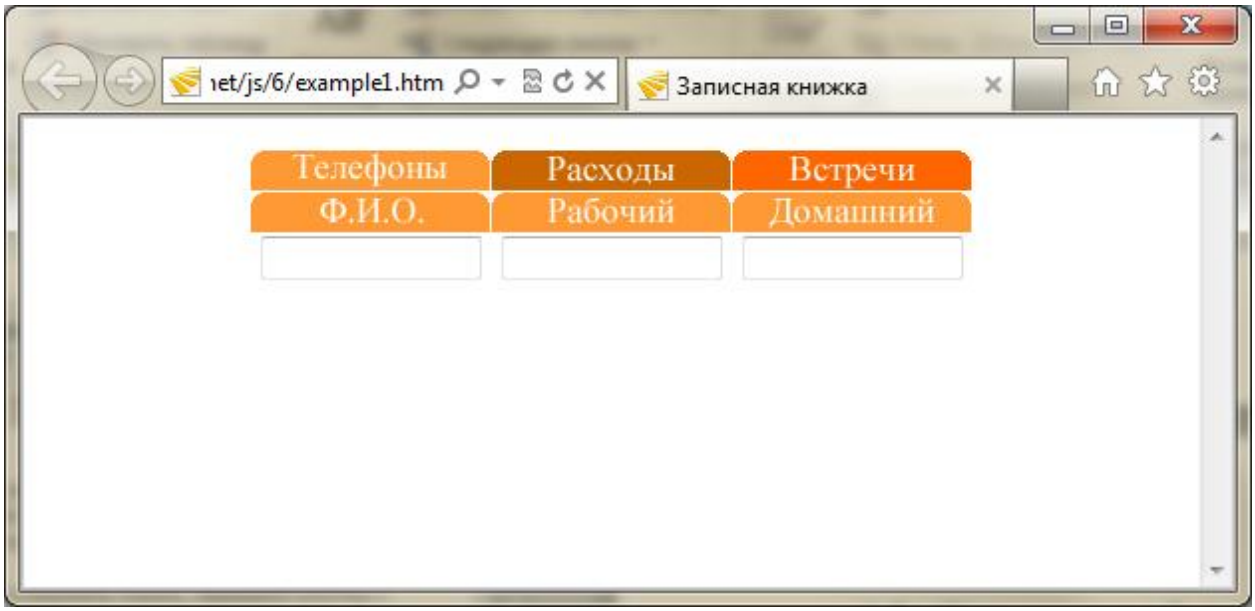
```
http://its.kpi.ua/help/index.html
http://its.kpi.ua/help/terms.html
http://its.kpi.ua/help/shop.html
javascript:alert('Область 1');
javascript:alert('Область 2');
```

Две новые ссылки — это ссылки из контейнера MAP, который не отображается, но ссылки из него попадают в коллекцию ссылок links[]. При этом они могут попасть между обычными гипертекстовыми ссылками, если контейнер MAP расположить внутри текста документа. Итак, ссылки, создаваемые контейнерами <A HREF="..."> и <AREA HREF="...">, равноправно присутствуют в коллекции document.links[].

6.3 Замена атрибута HREF

Выше мы перечислили свойства объекта класса URL. Теперь покажем, как при помощи **JavaScript**-кода можно этими свойствами управлять.

Пример **6.1.**




```
<html>
<head>

<title>Записная книжка</title>

<script>

var data = new Array(
"Фамилия И.О.",          "253-93-10",       "253-93-12",
"проезд",          "5",               "руб.",
"отчет",           "10:00",           "конф. зал.");

var pic = new Array(
"fio.gif",   "rpho.gif",  "hpho.gif",
"kind.gif",  "sum.gif",   "curr.gif",
"target.gif",      "mtime.gif", "mplace.gif");

function ShowField(m,i)
{
 document.f.elements[i].value = data[3*m+i];
}

function ShowMenu(menu)
{
 document.f.reset();

  for(i=0;i<3;i++)
  {
   document.images[i+3].src = pic[ 3*menu+i ];
   document.links[i+3].href =
   "javascript:ShowField("+menu+","+i+");void(0);";
  }
```

```
}
</script>
</head>
<body onLoad="ShowMenu(0);">

<table border=0 cellspacing=1 cellpadding=0 align=center>
<tr>
<td><a href="javascript:ShowMenu(0);void(0);"><img src=addrpho.gif border=0></a></td>
<td><a href="javascript:ShowMenu(1);void(0);"><img src=cash.gif    border=0></a></td>
<td><a href="javascript:ShowMenu(2);void(0);"><img src=meets.gif   border=0></a></td>
</tr>
<tr>
<td><a href=""><img src=fio.gif  border=0></a></td>
<td><a href=""><img src=rpho.gif border=0></a></td>
<td><a href=""><img src=hpho.gif border=0></a></td>
</tr>
<form name=f >
<th><input value="" size=14></th>
<th><input value="" size=14></th>
<th><input value="" size=14></th>
</form>
</table>

</body>
</html>
```

Рассмотрим меню типа "записная книжка". Конечно, это не настоящая записная книжка. Поле формы заполняется только при выборе гипертекстовой ссылки, расположенной над этим полем. Единственная цель данного примера — показать, как изменяется значение атрибута `HREF` (оно отображается в поле `status` окна браузера при наведении указателя мыши на ссылку). В этом примере изменение свойства `href` у ссылок производится точно так же, как изменение свойства `src` у картинок и свойства `value` у полей ввода в форме:

```
document.links[i+3].href = ...
document.images[i+3].src = ...
document.f.elements[i].value = ...
```

На нашей странице имеется 6 ссылок, 6 картинок и 3 поля ввода. Первые три ссылки и картинки (с номерами 0,1,2) всегда неизменны. Последние же три ссылки и картинки (т.е. с номерами 3,4,5) мы меняем вышеприведенными операторами (поэтому в них индекс `i+3`, где `i=0,1,2`). Как вы видите, мы меняем значение `href` целиком. Однако URL можно менять и частично.

6.4 Изменение части URL

Гипертекстовая ссылка — это объект класса `URL`, у которого помимо свойства `href` есть и другие, соответствующие частям URL, и их тоже можно менять. Проиллюстрируем эту возможность. Допустим, у нас имеется ссылка:

```
<A HREF="http://its.kpi.ua/courses/internet/js/">Курс по JavaScript</A>
```

Проверьте, что значение свойства `document.links[0].pathname` сейчас равно "`courses/internet/js/`". Применим оператор:

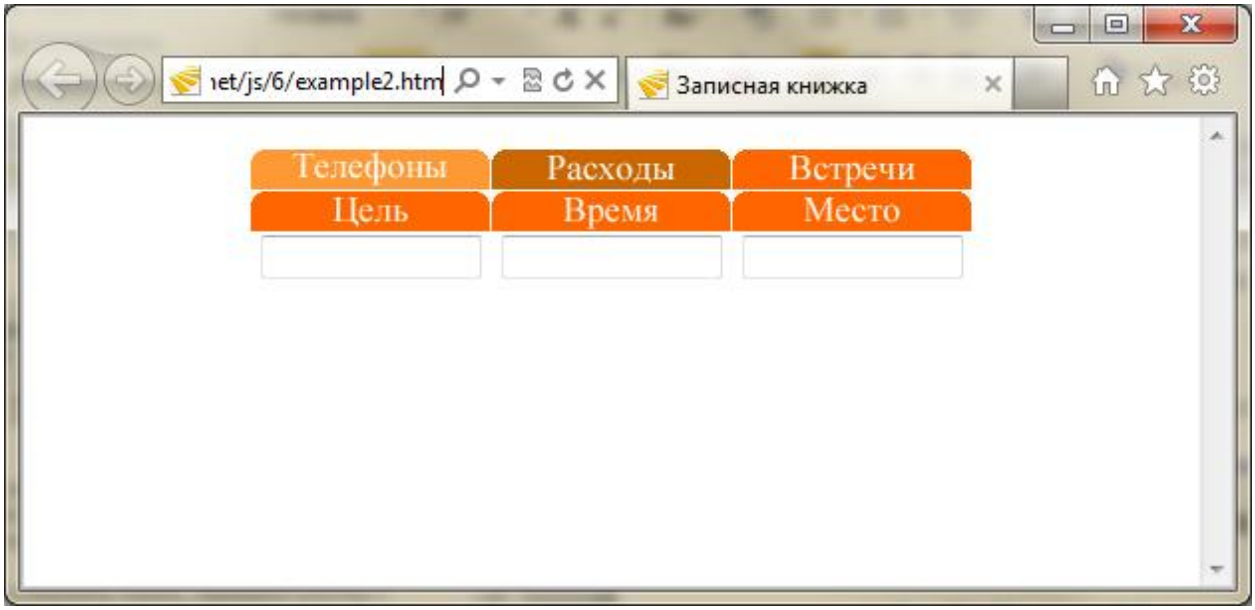
```
document.links[0].pathname="news/"
```

Теперь эта ссылка указывает на адрес `http://its.kpi.ua/news/`.

6.5 События MouseOver и MouseOut

Эти два события позволяют совершать действия при наведении или уводе курсора мыши на объекты, например, обесцвечивать и проявлять картинки, менять содержимое поля `status` и т.п. Применительно к гипертекстовым ссылкам, первое событие генерируется браузером, если курсор мыши находится над ссылкой, а второе — когда он покидает ссылку.

Пример **6.2.**




```
<html>
<head>
<title>Записная книжка</title>
<script>

var data = new Array(
"Фамилия И.О.",          "253-93-10",       "253-93-12",
"проезд",          "5",              "руб.",
```

```
"отчет",           "10:00",           "конф. зал.");

var pic = new Array(
"fio.gif",   "rpho.gif",  "hpho.gif",
"kind.gif",  "sum.gif",   "curr.gif",
"target.gif",      "mtime.gif", "mplace.gif");

function ShowField(m,i)
{
 document.f.elements[i].value = data[3*m+i];
}

function ShowMenu(menu)
{
 document.f.reset();

  for(i=0;i<3;i++)
  {
   document.images[i+3].src = pic[ 3*menu+i ];
   document.links[i+3].href =
   "javascript:ShowField("+menu+","+i+");void(0);";
  }
}
</script>
</head>
<body onLoad="ShowMenu(0);">

<table border=0 cellspacing=1 cellpadding=0 align=center>
<tr>
<td><a href="javascript:void(0);" onMouseOver="ShowMenu(0);"><img src=addrpho.gif
border=0></a></td>
<td><a href="javascript:void(0);" onMouseOver="ShowMenu(1);"><img src=cash.gif
border=0></a></td>
<td><a href="javascript:void(0);" onMouseOver="ShowMenu(2);"><img src=meets.gif
border=0></a></td>
</tr>
<tr>
<td><a href=""><img src=fio.gif  border=0></a></td>
<td><a href=""><img src=rpho.gif border=0></a></td>
<td><a href=""><img src=hpho.gif border=0></a></td>
</tr>
<form name=f >
<th><input value="" size=14></th>
<th><input value="" size=14></th>
<th><input value="" size=14></th>
</form>
</table>

</body>
</html>
```

Рассмотрим пример с записной книжкой, но теперь для появления подменю будем использовать обработчик события `onMouseOver`. Для этого достаточно заменить строчки вида:

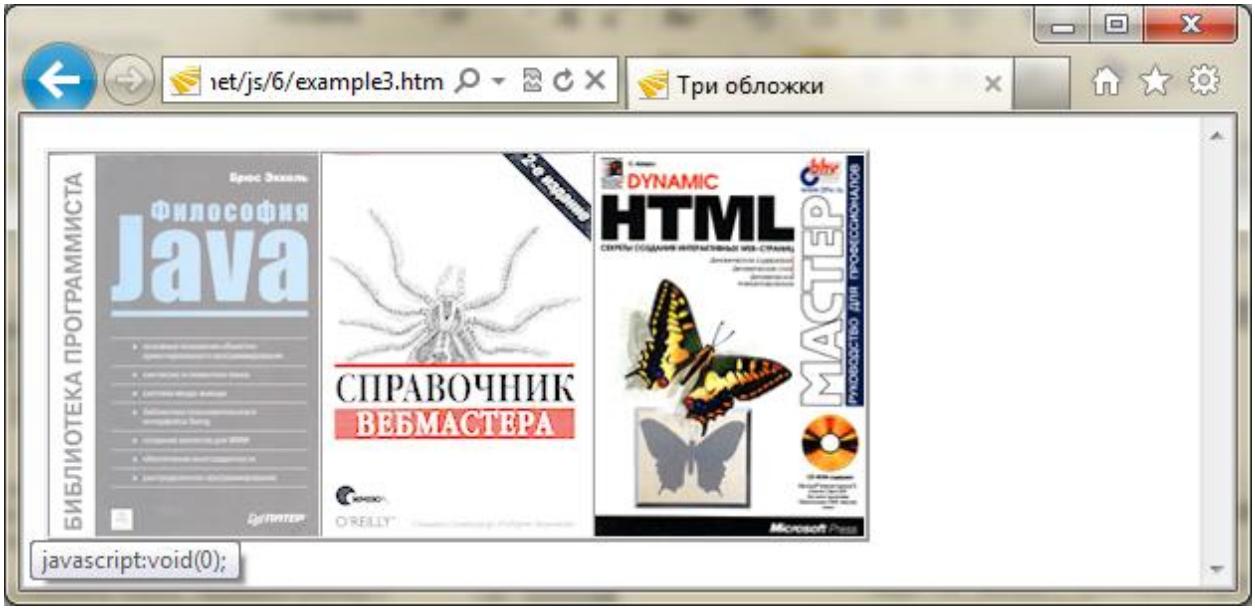
```
<A HREF="javascript:ShowMenu(0);void(0);">...</A>
```

открывавшие подменю, когда пользователь кликал по ссылке, на строчки вида:

```
<td><a href="javascript:void(0);" onMouseOver="ShowMenu(0);">...</A>
```

открывающие подменю при наведении указателя мыши на пункт меню верхнего уровня. Выражения "`javascript:void(0);`" мы оставили в атрибуте `HREF`, чтобы ничего не происходило, если пользователь случайно кликнет по пункту меню верхнего уровня.

Пример **6.3.**




```
<html>
<head>
<title>Три обложки</title>
</head>
<body>

<table border="2" cellpadding="0" cellspacing="0">
<tr>
<td>
<A HREF="javascript:void(0);"
 onMouseover="document.pic1.src='pic1.gif';"
 onMouseout="document.pic1.src='pic1_.gif';">
 <IMG NAME=pic1 src=pic1_.gif BORDER=0></A></td>
<td>
<A HREF="javascript:void(0);"
 onMouseover="document.pic2.src='pic2.gif';"
 onMouseout="document.pic2.src='pic2_.gif';">
 <IMG NAME=pic2 src=pic2_.gif BORDER=0></A></td>
<td>
<A HREF="javascript:void(0);"
 onMouseover="document.pic3.src='pic3.gif';"
 onMouseout="document.pic3.src='pic3_.gif';">
 <IMG NAME=pic3 src=pic3_.gif BORDER=0></A></td>
</tr>
</table>

</body>
</html>
```

В предыдущем примере нам не требовалось совершать какие-то действия при уводе указателя мыши с пункта меню. Теперь рассмотрим пример, когда это требуется, и для этого мы будем использовать обработчик события onMouseOut. При наведении указателя мыши на ссылку мы будем подменять картинку, а при уводе указателя мыши с картинки — восстанавливать ее. В исходном **HTML**-документе это будет выглядеть следующим образом:

```
<A HREF="javascript:void(0);"
 onMouseOver="document.pic1.src='pic1.gif';"
 onMouseOut="document.pic1.src='pic1_.gif';">
 <IMG NAME=pic1 src=pic1_.gif BORDER=0></A>
```

Как уже рассказывалось в лекции **4**, при возникновении события MouseOver у гиперссылки браузер показывает **URL** этой ссылки в поле статуса, а при возникновении события MouseOut восстанавливает в поле статуса прежнюю надпись. Перехватить первое событие (например, отменить вывод **URL** в строке статуса) можно, указав в его обработчике return true. Перехватить второе событие невозможно.

Примечание. В настоящее время обработчики событий onMouseOver и onMouseOut работают уже не только с гипертекстовыми ссылками, но и с изображениями, полями ввода, таблицами, их строками и ячейками, кнопками и другими **HTML**-элементами.

6.6 URL-схема "JavaScript:"

Для программирования гипертекстовых переходов в спецификацию **URL** разработчики **JavaScript** ввели, по аналогии с **URL**-схемами http, ftp и т.п., отдельную **URL**-схему javascript. Она уже упоминалась во вводной лекции при обсуждении вопроса о том, куда можно поместить **JavaScript**-программу в **HTML**-документе. Здесь мы рассмотрим ее подробнее.

Схема **URL** javascript: используется следующим образом:

```
<A HREF="JavaScript:код_программы">...</A>
<FORM ACTION="JavaScript:код_программы" ...> ... </FORM>
```

Рассмотрим пример гипертекстовой ссылки, в **URL** которой использована схема javascript:

```
<A HREF="javascript:alert('Спасибо!');">Кликните</A>
```

Как видим, если кликнуть по ссылке, то вместо перехода по какому-либо адресу просто появляется окно предупреждения.

В общем случае, после "javascript:" может стоять произвольная программа **JavaScript**. Что в этом случае будет происходить, если кликнуть такую ссылку? Ответ следующий: сначала **JavaScript**-программа будет исполнена, в результате чего будет вычислено ее *значение* (которым всегда считается значение **последнего** оператора в программе). Затем, если это значение неопределено (undefined), то далее ничего не произойдет; если же полученное значение определено, то на экран будет выведена **HTML**-страница с этим результатом (преобразованным в строку и воспринятым как **HTML**-документ, который в том числе может содержать и **HTML**-тэги).

В примере выше метод alert() не возвращает никакого значения, поэтому после появления окна предупреждения более ничего не происходит. Рассмотрим другие примеры. Кликнув по ссылке

```
<A HREF="javascript: 2+8; 5+7;">ссылка</A>
```

мы получим страницу, на которой написан результат последнего выражения 12 (первое выражение 2+8, конечно, тоже будет вычислено, но его результат никуда не пойдет). Если же кликнуть по такой ссылке:

```
<A HREF="javascript:
'<HTML><BODY><H1>Ready!<H1></BODY></HTML>';">Кликните</A>
```

то в окне браузера откроется страница, исходный **HTML**-код которой есть <HTML><BODY><H1>Ready!<H1></BODY></HTML>. Если нужно, чтобы при клике по ссылке просто выполнились какие-то действия и более ничего не происходило (в частности, не происходил переход к какой-либо другой странице), то в конце программы можно поместить оператор void(0) либо void(выражение). Эта функция вычисляет переданное в нее выражение и не возвращает никакого значения. Пример:

```
<A HREF="javascript: document.bgColor='green'; void(0);">Кликните</A>
```

Если кликнуть по такой ссылке, то изменится цвет фона, и больше ничего не произойдет. Без void(0) нам бы открылась **web**-страница со словом green.

Все сказанное справедливо и для использования **URL**-схемы "javascript:" в атрибуте ACTION контейнера <FORM>, за одним нюансом: в этом случае необходимо, чтобы в качестве метода доступа был указан METHOD="POST". Дело в том, что при использовании метода GET при отправке данных формы (т.е. при возникновении события Submit) к адресу **URL**, указанному в атрибуте ACTION, добавляется строка "?имя=значение&имя=значение&...", составленная из имен и значений элементов формы. Конечно же, такое выражение не является корректной **JavaScript**-программой, и браузеры могут выдавать сообщения об ошибке (а могут и не выдавать, а просто не выполнять программу). Пример:

```
<FORM NAME=f METHOD=post
ACTION="javascript:alert('Длина строки = '+document.f.e.value.length);">
<INPUT NAME=e>
<INPUT TYPE=submit VALUE="Длина">
</FORM>
```

Если ввести любую строку в поле ввода и нажать кнопку, то будет выведена длина строки. Этот пример был создан лишь для демонстрации **URL**-схемы "javascript:"; он весьма искусственный, т.к. в данном случае более уместно было бы использовать обработчик события onClick или onSubmit. Об этом и других применениях **JavaScript** в обработке *форм* рассказывается в лекции, посвященной программированию форм.

6.7 Обработка события Click

У гипертекстовой ссылки помимо **URL**, указанного в атрибуте HREF, можно указать действия, которые браузер должен выполнить, когда пользователь кликнет по данной ссылке, перед тем, как перейти по указанному **URL**. Соответствующая программа **JavaScript** называется *обработчиком события* Click и помещается в атрибут onClick контейнера <A>. Если обработчик события возвращает значение false (это можно реализовать путем помещения в конец обработчика команды return false), то переход по адресу **URL**, указанному в атрибуте HREF, не будет совершен. Если же обработчик

возвращает true либо ничего не возвращает, то после выполнения обработчика события будет совершен переход по адресу **URL**. Например:

```
<A onClick="return confirm('Хотите посетить сайт INTUIT?')"
  HREF="http://www.intuit.ru/">Перейти на сайт INTUIT</A>
```

В этом примере confirm() возвращает либо true, либо false, в зависимости от того, на какую кнопку нажмет пользователь в предложенном запросе. Соответственно, переход на указанный адрес либо произойдет, либо нет. Если бы мы в этом примере опустили слово return, то обработчик события ничего бы не возвращал (независимо от действий пользователя на запрос confirm) и переход на указанный **URL** совершался бы в любом случае.

Если в атрибуте ссылки (например, HREF, onClick и т.п.) пишется **JavaScript**-код, в котором надо сослаться на свойство или метод **этой** ссылки, то, как и в случае форм, можно пользоваться сокращенной записью — не указывать объект данной ссылки, либо (для большей ясности кода) вместо него писать this. Например, пусть у нас имеется **5**-я ссылка в документе, и мы хотим в ее обработчике onMouseOver сослаться на свойство href данной ссылки, или вызвать метод click() данной ссылки. Тогда в этом контексте вместо document.links[5].href можно писать this.href или просто href, а вместо document.links[5].click() писать this.click() или просто click(). Это не только укорачивает код, но и избавляет нас от необходимости привязываться к конкретному номеру данной ссылки, который в любой момент может измениться.

И последнее замечание. Часто для того, чтобы скрипт запускался, когда посетитель кликает ссылку, программисты пишут что-то такое:

```
<A HREF="#" onClick="программа JavaScript">...</A>
<A HREF="javascript:void(0)" onClick="программа JavaScript">...</A>
<A HREF="javascript: программа JavaScript">...</A>
```

При этом код выполняется и, на первый взгляд, все нормально. Но на второй взгляд становится видно, что после клика на ссылку могут прекратить грузиться недогруженные элементы страницы (большие картинки и т.п.), останавливаются анимированные **GIF**'ы и, быть может, происходит что-то еще из этой серии. Все дело тут в том, что браузер считает клик пользователя по ссылке переходом на другую страницу, поэтому полагает, что заботиться о текущей странице больше не надо, ведь она с секунды на секунду заменится новой. Решение здесь такое: если вы рассчитываете при клике пользователя по ссылке **оставить** его на текущей странице, то не забудьте прописать выход return false из обработчика события onClick, например:

```
<a href="#" onClick="программа JavaScript; return false;">
```

В этом случае после вызова программы **JavaScript** выполнение клика по ссылке прекратится, поэтому браузер не будет считать, что произошел переход на другую страницу.

7 ЛЕКЦИЯ: ПРОГРАММИРУЕМ ГРАФИКУ

Подробно рассказано о приемах программирования изменений графических образов на HTML-страницах, в частности, JavaScript-мультипликации и графических меню.

7.1 Объект Image

Наиболее зрелищные эффекты при программировании на **JavaScript** достигаются при работе с *графикой*. При этом в арсенале программиста не так уж много инструментов: встроенные в документ картинки, возможность генерации объекта `Image`, комбинирование картинок с гипертекстовыми ссылками и таблицами. Тем не менее обилие различных эффектов, которые достигаются этими нехитрыми средствами, впечатляет.

Программирование графики в **JavaScript** опирается на объект `Image`, который характеризуется следующими свойствами, методами и событиями:

Характеристики объекта `Image`		
Свойства	**Методы**	**События**

`name`
`src`
`lowSrc`
 	
`border`
`height`
`width`
`hspace`
`vspace`
 	
`complete`
	нет	`Abort`
		`Error`
		`Load`

Несмотря на такое обилие свойств, их абсолютное большинство можно только читать, но не изменять. Об этом свидетельствует, прежде всего, отсутствие методов. Но два свойства все же можно изменять: `src` и `lowSrc`. Этого оказывается достаточно для множества эффектов с картинками.

Все объекты класса `Image` можно разделить на встроенные и порожденные программистом. Встроенные объекты — это картинки контейнеров `IMG`. Если эти картинки поименовать, к ним можно обращаться по имени. Например, если у нас имеется картинка (будем считать, что она первая в документе):

```
<IMG NAME=picname SRC=forest.gif>
```

то значение свойства `document.images[0].name` будет равно "`picname`", а к самой картинке можно будет обращаться тремя способами:

```
document.images[0]
document.picname
document.images['picname']
```

7.1.1 Свойства src и lowSrc

Свойства `src` и `lowSrc` определяют **URL** изображения, которое монтируется внутрь документа. При этом `lowSrc` определяет временное изображение, обычно маленькое, которое отображается, пока загружается основное изображение, чей **URL** указывается в атрибуте `SRC` контейнера `IMG`. Свойство `src` принимает значение атрибута `SRC` контейнера `IMG`. Программист может изменять значения и `src`, и `lowSrc`. В предыдущем примере мы можем изменить значение `src` следующим образом:

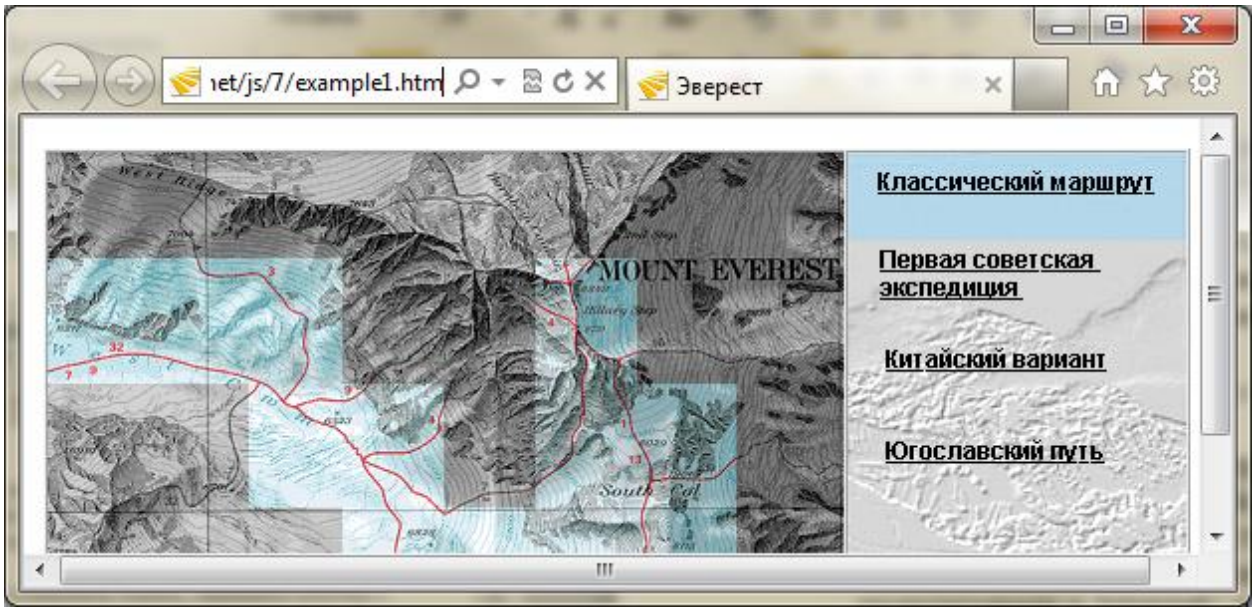
```
document.picname.src='river.gif';
```

Как видно из этого примера, существует возможность модифицировать картинку за счет изменения значения свойства `src` встроенного объекта `Image`. Если вы в первый раз просматриваете данную страницу (т.е. картинки не закешированы браузером), то постепенное изменение картинки будет заметно (конечно, при низкой скорости подключения к **Internet**; а также это зависит от браузера, который может загрузить картинку, а только потом вывести ее целиком на страницу). Как ускорить это изменение, мы рассмотрим в следующем разделе.

7.2 Изменение картинки

Изменить картинку можно, только присвоив свойству `src` встроенного объекта `Image` новое значение. Выше было показано, как это делается в простейшем случае. Очевидно, что медленная перезагрузка картинки с сервера не позволяет реализовать быстрое листание. Попробуем решить эту проблему. (Последующее в этом параграфе писалось в **2001–2002** годах, когда время подгрузки графики было значительным, чтобы можно было наблюдать описываемые ниже эффекты от оптимизации. — Прим.ред.).

Пример **7.1**




```
<HTML>
```

```
<HEAD>
<TITLE>Эверест</TITLE>
<SCRIPT>

var LastShownTrassa;

rm = new Array();
rc = new Array();

for(i=0;i<5;i++)
{
 rm[i] = new Image();
 rc[i] = new Image();

 rm[i].src = "rout00"+i+".gif";
 rc[i].src = "crou00"+i+".gif";
}

mono  = new Array(32);
color = new Array(32);

for(i=0;i<32;i++)
 {
    mono[i]  = new Image();
    color[i] = new Image();

    s = (i<10)? '0':'';

    mono[i].src  = "mapb0"+s+i+".gif";
    color[i].src = "mape0"+s+i+".gif";
 }

var trassa = new Array();

// Классический маршрут
trassa[0] = new Array(8,9,10,18,19,27,28,29,30,22,21,13);

// Первая советская экспедиция
trassa[1] = new Array(8,9,10,18,19,11,12,13);

// Китайский вариант
trassa[2] = new Array(6,5,13);

// Югославский путь
trassa[3] = new Array(0,1,2,3,4,12,13);

// Французский маршрут
trassa[4] = new Array(8,9,10,18,19,20,12,13);


function ChangeMap(who,colorit)
{ // Если colorit=true, то who-й маршрут (who<5)
  // закрасить цветом, иначе - черно-белым

  t=trassa[who];
  cvetkletki = (colorit)? "color" : "mono";
  cvetmenu   = (colorit)? "rc"    : "rm";

  for(k=0;k<t.length;k++)
  {
   kletka=t[k];
   eval("document.m"+kletka+".src="+cvetkletki+"["+kletka+"].src");
```

```
  }

 eval("document.r"+who+".src="+cvetmenu+"["+who+"].src");

 LastShownTrassa = (colorit)? who : 5;
}

function CleanMap()
{
 if(LastShownTrassa<5)             // Если еще не чистили старый маршрут, то
 ChangeMap(LastShownTrassa,false); // очистить LastShownTrassa-й маршрут
}

function ShowTrassa(NewTrassa)
{
 CleanMap();                 // Очистить карту
 ChangeMap(NewTrassa,true); // Проложить новый маршрут
}
</SCRIPT>

</HEAD>
<BODY>

<table border=1 cellspacing=0 cellpadding=0 width=540>
<tr><td>
<table border=0 cellspacing=0 cellpadding=0 width=400>
<script>
 d=""; // Составляем таблицу из 4x8 изображений

for(i=0;i<4;i++)
{
 d+="<tr>";      // начинаем генерировать ряд

 for(j=0;j<8;j++)
 {
  n=8*i+j; s= (n<10)? '0':'';
  d+=("<td><img name=m"+n+" src='mapb0"+s+n+".gif' border=0 "+
  "onMouseOver='document.m"+n+".src=color["+n+"].src;' "+
   "onMouseOut='document.m"+n+".src= mono["+n+"].src;'></td>");
 }
 d+="</tr>";    // конец ряда
}

document.write(d);

with(document) // подписи к рисункам
{
 m8.alt  = "Isefall Khumbu";
 m8.alt  = "Isefall Khumbu";
 m10.alt = "Gletcher Khumbu";
 m12.alt = "South Wall";
 m13.alt = "Mnt.Everest";
 m18.alt = "Gletcher Khumbu";
 m19.alt = "Gletcher Khumbu";
 m20.alt = "South Wall";
 m21.alt = "South Wall";
 m27.alt = "Gletcher Khumbu";
 m28.alt = "Gletcher Khumbu"
 m29.alt = "Pass";
 m30.alt = "Pass";
 m31.alt = "Pass";
}
```

```
</script>

</table>
</td>

<td>
<table border=0 cellspacing=0 cellpadding=0>

<script>
for(i=0;i<5;i++)  // Составляем правое меню
document.write(
'<tr><td><img name=r'+i+' src="rout00'+i+'.gif" border=0 '+
'onMouseOver="ShowTrassa('+i+');" '+
'onMouseOut="CleanMap();"></td></tr>');
</script>

</table>

</td></tr>
</table>

</BODY>
</HTML>
```

Решение заключается в разведении по времени подкачки картинки и ее отображения. Для начала мы создаем изображения, к которым привязываем обработчики событий `onMouseOver` и `onMouseOut`. При наведении указателя мыши на каждую из картинок она заменяется другой (цветной), а при уводе мышки картинка заменяется обратно на черно-белую:

```
<IMG NAME=m0 src="images/mapb000.gif" border=0
   onMouseOver="document.m0.src=color[0].src;"
    onMouseOut="document.m0.src= mono[0].src;">
```

Более того, если навести мышку на пункт меню справа от карты, то вызовется функция, которая заменяет адреса сразу у нескольких картинок.

Однако, главное не в том, что картинки замещаются, а в том, с какой скоростью они это делают. Для достижения нужного результата в начале страницы создаются массивы картинок, в которые перед отображением перекачивается вся графика. Для этой цели используют конструктор объекта `Image`:

```
mono = new Array(32);
for(i=0;i<32;i++)
{
 mono[i]  = new Image();
 s = (i<10)? "0" : "";
 mono[i].src  = "images/mapb0" +s+i+ ".gif";
}
```

Именно в тот момент, когда свойству, например, `mono[25].src` присваивается значение "`images/mapb025.gif`", и происходит скачивание этой картинки с сайта на компьютер пользователя. Если бы вместо объектов класса `Image` мы составили массив из строк вида "`images/mapb000.gif`" и т.д., то никакой подгрузки графики не произошло бы, и каждый раз, когда пользователь наводил бы на очередное изображение, браузеру приходилось бы скачивать новую картинку. Мы поместили этот скрипт в контейнер `<HEAD>`, тем самым гарантировав, что к моменту, когда пользователь начнет работать со

страницей, все требуемые для работы картинки уже будут скачаны, и в процессе вождения мышки по картинкам никакой задержки показа очередного изображения наблюдаться не будет.

Попутно обратим внимание на следующий трюк, использованный в скрипте. Предположим, нам необходимо написать **32** строчки:

```
document.m0.src = mono[0].src;
document.m1.src = mono[1].src;
...
document.m31.src = mono[31].src;
```

Мы хотим избежать написания такого громоздкого кода. С правой частью операторов присваивания справиться легко: достаточно задать цикл по `i` от `0` до `31` и писать `mono[i].src`. А вот с левой частью не все так просто. На помощь приходит функция `eval("выражение")`, которая воспринимает переданное ей `выражение` как программу **JavaScript** и выполняет все содержащиеся в ней операторы. С ее помощью решить нашу проблему легко — мы составляем нужную строчку, а затем отдаем ее на выполнение:

```
for(i=0;i<32;i++)
eval("document.m" +i+ ".src = mono[" +i+ "].src;");
```

7.3 Мультипликация

Естественным продолжением идеи замещения значения атрибута `SRC` в контейнере `IMG` является *мультипликация*, т.е. последовательное изменение значения этого атрибута во времени. Для реализации мультипликации используют метод `setTimeout()` объекта `window` (см. лекцию **4**).

Существует два способа запуска мультипликации: по окончании загрузки страницы (`onLoad`) и при действиях пользователя (`onClick`, `onChange` и т.д.). Наиболее популярный — первый, а именно использование `onLoad()` и `setTimeout()`.

7.3.1 Обработчик события onLoad

Событие `Load` наступает в момент окончания загрузки документа браузером. Обработчик данного события (`onLoad`) указывается в контейнере `BODY`:

```
<BODY onLoad="программа JavaScript">
```

Рассмотрим сначала пример, в котором при загрузке документа начинает выполняться бесконечный цикл изменения картинки:

```
<SCRIPT>
var i=0;
function movie()
{
 document.i.src='images/crou00'+i+'.gif';
 i++; if(i>4) i=0;
 setTimeout('movie();',500);
}
</SCRIPT>
<BODY onLoad="movie();">
<IMG NAME=i>
</BODY>
```

Пример 7.1. Бесконечная мультипликация

Можно реализовать и конечное число циклов мультипликации, скажем 5:

```
<SCRIPT>
var i=0,
    n=5; // число циклов
function movie()
{
 document.i.src='images/crou00'+i+'.gif';
 i++; if(i>4) { i=0; n--; }
 if(n>0) setTimeout('movie();',500);
}
</SCRIPT>
<BODY onLoad="movie();">
<IMG NAME=i>
</BODY>
```

Пример 7.2. Мультипликация с числом циклов n=5

В обоих примерах следует обратить внимание на использование метода setTimeout(). На первый взгляд, это просто рекурсия. Но в действительности все несколько сложнее. **JavaScript** разрабатывался для многопоточных операционных систем, поэтому правильнее будет представлять себе исполнение скриптов следующим образом:

1. скрипт movie() получает управление от обработчика события onLoad;
2. заменяет картинку;
3. порождает новый скрипт movie() и откладывает его исполнение на **500** миллисекунд;
4. текущий скрипт movie() уничтожается **JavaScript**-интерпретатором.

После окончания срока задержки исполнения все повторяется. В первом примере (бесконечное повторение) функция порождает саму себя и, тем самым, поддерживает непрерывность своего выполнения. Во втором примере (конечное число итераций) после нескольких повторов функция не порождается. Это приводит к завершению процесса отображения новых картинок.

7.3.2 Запуск и остановка мультипликации

Постоянная мультипликация может быть достигнута и другими средствами, например многокадровыми графическими файлами. Однако движение на странице — не всегда благо. Часто возникает желание реализовать запуск и остановить движения по требованию пользователя. Удовлетворим это желание, модифицировав пример с бесконечной мультипликацией:

```
<SCRIPT>
var i=0, flag=true;
function movie()
{
 if(flag)
 {
  document.i.src='images/crou00'+i+'.gif';
  i++; if(i>4) i=0;
 }
 setTimeout('movie();',500);
}
</SCRIPT>
<BODY onLoad="movie();">
<FORM>
<INPUT TYPE=button VALUE="Start/Stop"
       onClick="flag = !flag;">
```

```
</FORM>
<IMG NAME=i>
</BODY>
```

Пример 7.3. Остановка/запуск мультипликации (поток генерируется постоянно)

Мы ввели булевский `flag` и меняем изображение на странице, только если он принимает значение `true`. Нажатие на кнопку меняет значение флага на его отрицание.

Обратите внимание: когда `flag` имеет значение `false`, мы просто обходим изменение картинки, но не прекращаем порождение потока. Если бы мы поместили `setTimeout()` внутрь конструкции `if()`, то после нажатия на кнопку поток перестал бы порождаться, и запустить мультипликацию заново стало бы нельзя. Однако постоянное генерирование потока — это некоторая растрата ресурсов (памяти, процессора). Нельзя ли сделать оптимальнее? Можно. Этот способ основан на применении метода `clearTimeout()`:

```
<SCRIPT>
var i=0, flag=true, m=null;
function movie()
{
 if(flag)
 {
  document.i.src='images/crou00'+i+'.gif';
  i++; if(i>4) i=0;
 }
 m = setTimeout('movie();',500);
}
</SCRIPT>
<BODY onLoad="movie();">
<IMG NAME=i>
<FORM>
<INPUT TYPE=button VALUE="Start/Stop"
  onClick="flag = !flag;
  if(flag) movie();
  else clearTimeout(m);">
</FORM>
</BODY>
```

Пример 7.4. Остановка/запуск мультипликации (поток приостанавливается)

Как видите, достаточно ввести идентификатор потока `m` и сохранять в нем ссылку на поток при вызове `setTimeout()`. Тогда в случае необходимости (при нажатии пользователем кнопки) мы можем отменить запланированное выполнение `movie()` (которое произошло бы через **500** миллисекунд), вызвав метод `clearTimeout(m)`.

7.4 Оптимизация отображения

При программировании графики следует учитывать множество факторов, которые влияют на скорость отображения страницы и скорость изменения графических образов. При этом обычная дилемма оптимизации программ — скорость или размер занимаемой памяти — решается только в пользу увеличения скорости. О размере памяти при программировании на **JavaScript** думать как-то не принято.

Из всех способов оптимизации отображения картинок мы остановимся только на нескольких:

- оптимизация отображения при загрузке;
- оптимизация отображения за счет предварительной загрузки;
- оптимизация отображения за счет нарезки изображения.

Если первые две позиции относятся в равной степени как к отображению статических картинок, так и к мультипликации, то третий пункт характерен главным образом для мультипликации.

7.4.1 Оптимизация при загрузке изображений

Практически в любом руководстве по разработке **HTML-**страниц отмечается, что при использовании контейнера `IMG` в теле **HTML-**страницы следует указывать атрибуты `WIDTH` и `HEIGHT`. Это продиктовано порядком загрузки компонентов страницы с сервера и алгоритмом работы **HTML-**парсера. Первым загружается текст разметки. После этого парсер разбирает текст и начинает загрузку дополнительных компонентов, в том числе графики. При этом загрузка картинок, в зависимости от типа **HTTP-**протокола, может идти последовательно или параллельно.

Также параллельно с загрузкой парсер продолжает свою работу. Если для картинок заданы параметры ширины и высоты, то можно отформатировать текст и отобразить его в окне браузера. До тех пор, пока эти параметры не определены, отображения текста не происходит. (В современных браузерах текст обычно отображается сразу, оставляя картинке "небольшое" место; по окончании загрузки картинки (или когда становятся известными ее размеры) страница переформатируется "на лету". — Прим.ред.)

Таким образом указание высоты и ширины картинки позволит отобразить документ раньше, чем картинки будут получены с сервера. Это дает пользователю возможность читать документ или задействовать его гипертекстовые ссылки до момента полной загрузки документа.

С точки зрения **JavaScript**, указание размеров картинки задает начальные параметры окна отображения графики внутри документа. Это позволяет воспользоваться маленьким прозрачным образом, для того, чтобы заменить его полноценной картинкой. Идея состоит в передаче маленького объекта для замещения его по требованию большим объектом.

7.4.2 Предварительная загрузка изображений

Замена одного образа другим часто бывает оправдана только в том случае, когда это происходит достаточно быстро. Если перезагрузка длится долго, то эффект теряется. Для быстрой подмены используют возможность предварительной загрузки документа в специально созданный объект класса `Image`.

Реальный эффект можно почувствовать только при отключении кэширования страниц на стороне клиента (браузера). Кэширование часто используют для ускорения работы со страницами **Web-**узла. Как правило, загрузка первой страницы — это достаточно длительный процесс. Самое главное, чтобы пользователь в этот момент был готов немного подождать. Поэтому, кроме графики, необходимой только на первой странице, ему можно передать и *графику*, которая на ней не отображается. Но зато при переходе к другим страницам узла она будет отображаться без задержки на передачу с сервера.

Описанный выше прием неоднозначен. Его оправдывает только то, что если пользователь нетерпелив, то он вообще отключит передачу графики.

7.4.3 Нарезка изображений

Нарезка картинок применяется довольно часто. Она позволяет достигать эффекта частичного изменения отображаемой картинки. Чаще всего он применяется при создании меню.

Кроме подобного эффекта нарезка позволяет реализовать мультипликацию на больших картинках. При этом изменяется не весь образ, а только отдельные его части.

7.5 Графика и таблицы

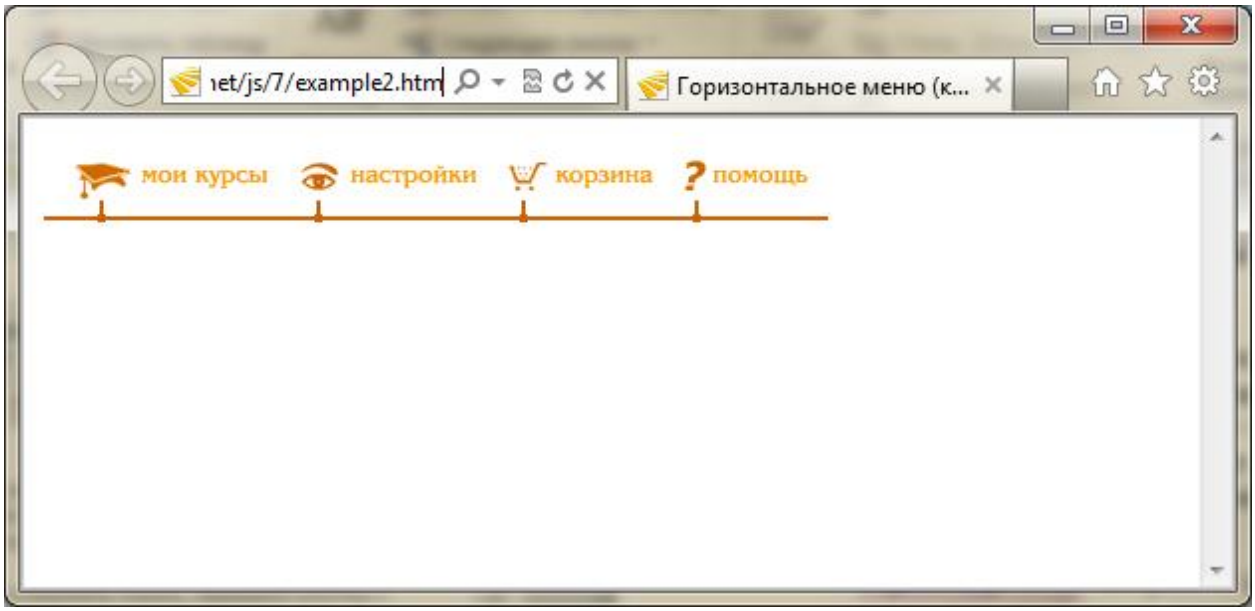
Одним из наиболее популярных приемов дизайна страниц **Web-**узла является техника нарезки картинок на составные части. Можно выделить следующие способы применения этой техники для организации навигационных компонентов страницы:

- горизонтальные и вертикальные меню;
- вложенные меню;
- навигационные графические блоки.

7.5.1 Горизонтальное меню

Главной проблемой при использовании нарезанной графики является защита ее от контекстного форматирования страницы **HTML-**парсером. Дело в том, что он автоматически переносит элементы разметки на новую строку, если они не помещаются в одной. Составные части нарезанной картинки должны быть расположены на экране определенным образом, но простое их перечисление в ряд не дает желаемого эффекта.

Пример **7.2**.



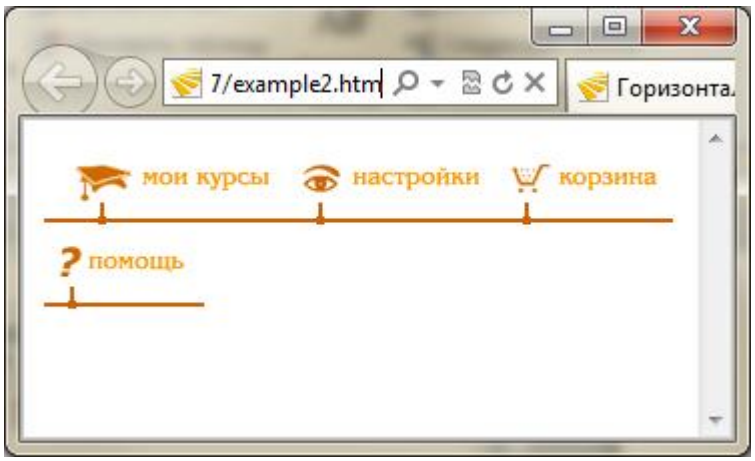


Рисунок 7.1 - Горизонтальное меню (картинки съезжают в узком окне)

```
<HTML>
<HEAD>
<TITLE>Горизонтальное меню (картинки съезжают в узком окне)</TITLE>
</HEAD>
<BODY>

<IMG SRC="horis1.gif">
<IMG SRC="horis2.gif">
<IMG SRC="horis3.gif">
<IMG SRC="horis4.gif">

</BODY>
</HTML>
```

Если открыть данный пример в новом окне и попробовать уменьшить ширину окна, так чтобы она стала меньше общей ширины всех картинок, то вы увидите, что картинки начинают переноситься на новую строку, когда ширина раздела становится меньше общей ширины всех картинок:

Проблема решается применением "защиты" от переноса на следующую строчку — контейнера <PRE>:

```
<PRE>
<IMG SRC="horis1.gif"><IMG
     SRC="horis2.gif"><IMG
     SRC="horis3.gif"><IMG
     SRC="horis4.gif">
</PRE>
```

Рис. 7.2. Горизонтальное меню (картинки защищены от переноса)

Если мы хотим использовать эту последовательность картинок как меню, то нужно задать гипертекстовые ссылки, что приводит к появлению рамок вокруг изображений. Это происходит потому, что по умолчанию толщина границ у изображений, являющихся ссылками — ненулевая.

```
<PRE>
<A HREF="courses.htm"><IMG SRC="horis1.gif"></A><A
```

```
   HREF="setting.htm"><IMG SRC="horis2.gif"></A><A
   HREF="baskets.htm"><IMG SRC="horis3.gif"></A><A
   HREF="thehelp.htm"><IMG SRC="horis4.gif"></A>
</PRE>
```

Пример 7.7. Горизонтальное меню (рамки вокруг картинок)

Рис. 7.3. Горизонтальное меню (рамки вокруг картинок)

Устранить этот недостаток можно путем задания значения атрибута `BORDER=0` у изображений:

```
<PRE>
<A HREF="courses.htm"><IMG SRC="horis1.gif" BORDER=0></A><A
   HREF="setting.htm"><IMG SRC="horis2.gif" BORDER=0></A><A
   HREF="baskets.htm"><IMG SRC="horis3.gif" BORDER=0></A><A
   HREF="thehelp.htm"><IMG SRC="horis4.gif" BORDER=0></A>
</PRE>
```

Пример 7.8. Горизонтальное меню (рамки более не видны)

Рис. 7.4. Горизонтальное меню (рамки более не видны)

7.5.2 Вертикальное меню

Теперь попробуем тем же способом реализовать вертикальное меню. При просмотре в некоторых браузерах сплошной вертикальной линии не получается, т.к. высота строки не равна высоте картинки. Подогнать эти параметры практически невозможно. Каждый пользователь настраивает браузер по своему вкусу.

```
<PRE>
<IMG SRC=vert.gif WIDTH=27 HEIGHT=21 BORDER=0><A
HREF="courses.htm"><IMG SRC="vert1.gif" WIDTH=103 HEIGHT=21 BORDER=0></A>
<IMG SRC=vert.gif WIDTH=27 HEIGHT=21 BORDER=0><A
HREF="setting.htm"><IMG SRC="vert2.gif" WIDTH=103 HEIGHT=21 BORDER=0></A>
<!-- далее аналогично для vert3.gif и vert4.gif -->
</PRE>
```

Пример 7.9. Вертикальное меню (линия не сплошная)

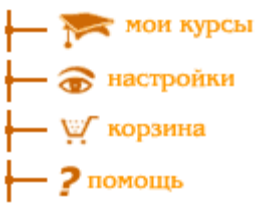


Рис. 7.5. Вертикальное меню (линия не сплошная)

Решение заключается в использовании (вместо контейнера <PRE>) таблицы <TABLE> с нулевыми границами между ячейками:

```
<TABLE BORDER=0 CELLPADDING=0 CELLSPACING=0>
<TR>
<TD><IMG SRC=vert.gif WIDTH=27 HEIGHT=21 BORDER=0></TD>
<TD><A HREF="courses.htm"><IMG SRC="vert1.gif"
       WIDTH=103 HEIGHT=21 BORDER=0></A></TD>
</TR>
<!-- далее аналогично для vert2, vert3, vert4.gif -->
</TABLE>
```

Пример 7.10. Вертикальное меню (линия сплошная)

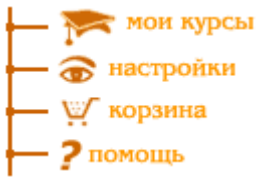


Рис. 7.6. Вертикальное меню **(**линия сплошная**)**

В данном случае все картинки удается сшить без пропусков и тем самым достичь непрерывности навигационного дерева. Пропуски устраняются путем применения атрибутов BORDER, CELLSPACING и CELLPADDING. Первый устраняет границы между ячейками и вокруг всей таблицы, второй устанавливает расстояние между ячейками равным **0** пикселов, третий устанавливает отступ между границей ячейки и элементом, помещенным в нее, в **0** пикселов.

7.5.3 Выделение выбранного пункта меню

Практически все, что было изложено до сих пор, касается вопросов построения одноуровневых меню. Поэтому в данном разделе мы постараемся привести более или менее реальные примеры таких меню. Графическое меню удобно тем, что автор может всегда достаточно точно расположить его компоненты на экране. Это, в свою очередь, позволяет и другие элементы страницы точнее располагать относительно элементов меню.

В данном примере мы воспользуемся этим для того, чтобы выделить тот пункт меню, над которым находится указатель мыши. Поскольку пункты меню прилегают друг к другу без просветов, то мы будем указывать стрелочкой тот пункт меню, который активен в данный момент.

```
<TABLE BORDER=0 CELLPADDING=0 CELLSPACING=0>
<TR ALIGN="center">

<TD>
<IMG NAME=e1 SRC=empty.gif WIDTH=15 HEIGHT=8 BORDER=0>
</TD>

<!-- аналогично для e2, e3, e4 -->

</TR>
<TR>

<TD><A HREF="javascript:void(0);"
 onMouseover="document.e1.src='arrowdw.gif';"
 onMouseout="document.e1.src='empty.gif';">
 <IMG SRC="horis1.gif" BORDER="0"></A>
</TD>

<!-- аналогично для e2, e3, e4 -->
```

```
</TR>
</TABLE>
```

Пример 7.11. Горизонтальное меню (пункт указан стрелкой)

Рис. 7.7. Горизонтальное меню (пункт указан стрелкой)

Стрелочка бежит точно над тем элементом, на который указывает мышь.

Однако стоит заметить, что применение атрибута ALT у контейнера IMG и его дублирование в строке статуса является гораздо более информативным, чем добавление нового графического элемента. Правда, отображается содержание ALT с некоторой задержкой:

```
<PRE>
<A HREF="courses.htm" onMouseOut="window.status='';"
   onMouseOver="window.status='Мои курсы';return true;"><IMG
   SRC="horis1.gif" BORDER=0 ALT="Мои курсы"></A><A

   HREF="setting.htm" onMouseOut="window.status='';"
   onMouseOver="window.status='Настройки';return true;"><IMG
   SRC="horis2.gif" BORDER=0 ALT="Настройки"></A><A

   HREF="baskets.htm" onMouseOut="window.status='';"
   onMouseOver="window.status='Корзина';return true;"><IMG
   SRC="horis3.gif" BORDER=0 ALT="Корзина"></A><A

   HREF="thehelp.htm" onMouseOut="window.status='';"
   onMouseOver="window.status='Помощь';return true;"><IMG
   SRC="horis4.gif" BORDER=0 ALT="Помощь"></A>
</PRE>
```

Пример 7.12. Горизонтальное меню (атрибут ALT и поле статуса)

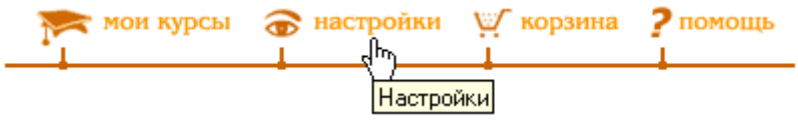


Рис. 7.8. Горизонтальное меню (атрибут **ALT** и поле статуса)

Посмотрим теперь на реализацию вертикального меню, построенного на основе графических блоков текста:

```
<TABLE BORDER=0 CELLPADDING=0 CELLSPACING=0>
<SCRIPT>
for(i=1; i<5; i++)
{ var
 ileft  = '<IMG SRC=block'+i+'.gif border=0>',
 iright = '<IMG NAME=e'+i+' SRC=clear.gif border=0>',
 isrc   = 'document.e'+i+'.src',
 alink  = '<A HREF="javascript:void(0);" '+
          'onMouseover="'+isrc+'=\'corner.gif\';" '+
          'onMouseout="'+ isrc+'=\'clear.gif\';">';

 document.write('<TR>'+
 '<TD>'+ alink + ileft  + '</A></TD>'+
```

```
 '<TD>'+ alink + iright + '</A></TD>'+
 '</TR>');
}
</SCRIPT>
</TABLE>
```

Пример 7.13. Вертикальное меню (графические блоки текста)

Здесь продемонстрировано типичное применение **JavaScript —** многократное генерирование похожих фрагментов **HTML-**кода с помощью цикла и метода `document.write()`**.** В данном примере с помощью цикла мы генерируем **HTML-**фрагмент следующего вида **(**каждый раз с разным номером вместо **"1"):**

```
<TR>
<TD><A onMouseover="document.e1.src='corner.gif';"
        onMouseout="document.e1.src='clear.gif';"
  HREF="javascript:void(0);"><IMG SRC=block1.gif border=0></A></TD>
<TD><A onMouseover="document.e1.src='corner.gif';"
        onMouseout="document.e1.src='clear.gif';"
 HREF="javascript:void(0);"><IMG NAME=e1 SRC=clear.gif border=0></A></TD>
</TR>
```

При движении мыши у соответствующего пункта меню**,** попавшего в фокус мыши**,** "отгибается уголок"**.** В данном случае "уголок" **—** это самостоятельная картинка**.** Все уголки реализованы в правой колонке таблицы**.** Для того чтобы гипертекстовая ссылка срабатывала по обеим картинкам **(**тексту и "уголку"**),** применяются одинаковые контейнеры `<A>`**,** охватывающие каждую картинку**.**

Рис. 7.9. Вертикальное меню **(**графические блоки текста**)**

7.6 Вложенные меню

В **HTML** нет стандартного способа реализации вложенных меню**.** Тем не менее за счет графики можно создать их подобие**.** При этом следует понимать**,** что место**,** на которое ложится графика**,** нельзя заполнить текстом**.**

В этом примере вложенное меню расположено справа от основного**.** Эффект вложенности достигается за счет изменения цвета **(**подпункты имеют цвет**,** отличный от цвета пунктов**).**

```
<SCRIPT>
function submenu(a)
{
 with(document)
 {
  if(a==1)
  {
   item1.src="item_1_yes.gif"; // 1-й пункт активен
   item2.src="item_2_no.gif";  // 2-й пункт неактивен
   subi1.src="item_1_1.gif";   // 1-й пункт вложенного меню 1
```

```
   subi2.src="item_1_2.gif";   // 2-й пункт вложенного меню 1
  }
  if(a==2)
  {
   item1.src="item_1_no.gif";  // 2-й пункт активен
   item2.src="item_2_yes.gif"; // 1-й пункт неактивен
   subi1.src="item_2_1.gif";   // 1-й пункт вложенного меню 2
   subi2.src="item_2_2.gif";   // 2-й пункт вложенного меню 2
  }
 }
}
</SCRIPT>

<TABLE BORDER=0 CELLPADDING=0 CELLSPACING=1>
<TR>
<TD><A HREF="javascript:void(0);" onMouseOver="submenu(1);">
    <IMG BORDER=0 NAME=item1 SRC=item_1_yes.gif></A></TD>
<TD><IMG BORDER=0 NAME=subi1 SRC=item_1_1.gif></TD></TR>
<TR>
<TD><A HREF="javascript:void(0);" onMouseOver="submenu(2);">
    <IMG BORDER=0 NAME=item2 SRC=item_2_no.gif></A></TD>
<TD><IMG BORDER=0 NAME=subi2 SRC=item_1_2.gif></TD></TR>
</TABLE>
```

Пример 7.14. Вложенное меню (пункт выделен цветом)

Рис. 7.10. Вложенное меню **(**пункт выделен цветом**)**

Подчиненность меню можно подчеркнуть изменением его положения относительно основного меню **(**составьте соответствующий код самостоятельно**):**

Рис. 7.11. Вложенное меню **(**пункт выделен сдвигом**)**

В этом случае для продвижения меню вниз необходимо зарезервировать место при помощи невидимых или видимых картинок**.**

При использовании слоев можно создать настоящее выпадающее меню**.**

8 ЛЕКЦИЯ: ПРОГРАММИРУЕМ "ЗА КАДРОМ"

Рассматриваются приемы программирования на JavaScript, невидимые для читателей HTML-страниц: механизм cookie, управление фокусом, скрытая передача данных, вопросы безопасности.

8.1 Механизм cookie

Cookie являются механизмом управления обменом данных. Основная их функция - поддержка сеанса работы между клиентом (браузером) и сервером.

8.1.1 Что такое cookie

Формально, ***cookie*** (читается: *куки*; не склоняется) — это небольшой фрагмент данных, которые веб-браузер пересылает веб-серверу в **HTTP**-запросе при каждой попытке открыть очередную страницу сайта. Обычно куки создаются веб-сервером и присылаются в браузер при первом запросе к сайту. Куки также могут быть созданы самой загруженной **web**-страницей, а именно имеющимся в ней скриптом **JavaScript**. Далее хранятся на компьютере пользователя в виде текстового файла, до тех пор, пока либо не закончится их срок, либо они будут удалены скриптом или пользователем. Имеются ограничения на объем и количество хранимых **cookie**, они зависят от браузера, но есть минимальные требования. Спецификация **cookie** описана в **RFC 2965**. На практике **cookie** обычно используются для:

- аутентификации пользователя;
- хранения персональных предпочтений и настроек пользователя;
- отслеживания состояния сессии доступа пользователя;
- ведения статистики о пользователях.

Без куки просмотр каждой веб-страницы является изолированным действием, не связанным с просмотром других страниц того же сайта. В конце этого раздела мы создадим страницы, передающие друг другу информацию с помощью куки.

Главными атрибутами **cookie** являются: имя/значение куки, срок действия, путь, доменное имя, шифрование. Атрибуты записываются через точку с запятой. Обязательным является лишь имя/значение. Например:

```
customer=21584563; expires=Fri, 31-Dec-2010 23:59:59 GMT;
path=/; domain=www.shop.ru; secure
```

Здесь `customer` — имя куки; `21584563` — значение куки; `expires=...` — срок действия куки; `path=/` — путь, для которого действует эта куки (в данном случае куки действуют для любой страницы на данном сайте); `domain=www.shop.ru` — домен (сайт), для которого будет действовать эта куки; `secure` — использовать ли для передачи куки шифрованный канал (**HTTPS**). Если не указан `expires`, то куки действительна до закрытия браузера. Если не указаны `path` и `domain`, то куки действительна для текущей **web**-страницы. Удаляются куки путем указания истекшего срока действия (браузер сам сотрет такие куки по окончании своей работы). Подробнее о формате **cookie** рассказано в лекции **9**. Здесь же мы остановимся на методах работы с **cookie** с помощью инструментов **JavaScript**.

8.1.2 Чтение cookie

Для работы с куки из сценария **JavaScript** используется свойство `document.cookie`. Следующая команда покажет все установленные куки:

```
alert(document.cookie);
```

Она выдаст нечто вроде: `name1=value1; name2=value2; ...`, т.е. перечисление пар `имя=значение`, разделенных точкой с запятой и пробелом. Как видите, нам показывают только имена и значения куки; другие атрибуты куки (срок действия, домен и т.д.) через свойство `document.cookie` недоступны.

Но обычно требуется больше — узнать, установлено ли значение конкретной куки, и если установлено, то прочитать его. Значит, нужно разобрать полученную выше строку с помощью методов работы со строковыми объектами. Для этого создадим две функции: `existsCookie` — проверяет, имеется ли куки с данным именем; `CookieValue` — возвращает значение куки по ее имени:

```
function existsCookie(CookieName)
{ // Узнает, имеется ли куки с данным именем
 return (document.cookie.split(CookieName+'=').length>1);
}

function CookieValue(CookieName)
{ // Выдает значение куки с данным именем
 var razrez = document.cookie.split(CookieName+'=');
 if(razrez.length>1)
 { // Значит, куки с этим именем существует
  var hvost = razrez[1],
       tzpt = hvost.indexOf(';'),
  EndOfValue = (tzpt>-1)? tzpt : hvost.length;
  return unescape(hvost.substring(0,EndOfValue));
}}
```

Мы воспользовались тем, что пары имя/значение разделены точкой с запятой, а значение куки не может содержать символ "точка с запятой" (";"). На самом деле, если произвести попытку установить куки со значением, содержащим этот символ, то в результате значение будет обрезано до первого вхождения этого символа. Попутно заметим, что в целях совместимости не рекомендуется использовать в значении куки символы: точка с запятой, пробел, равенство — если они все же требуются, их следует заменить на `%3B`, `%20` и `%3D`, соответственно. Проще всего при создании куки пользоваться функцией `escape()`, которая и произведет все эти преобразования, а при чтении куки — обратной функцией `unescape()`, что мы и сделали выше.

8.1.3 Создание или изменение cookie

Далее мы хотим создавать или менять **cookie**. Разработчики языка **JavaScript** позаботились о **web-**программистах и реализовали свойство `document.cookie` довольно интеллектуально. Если выполнить простую команду присвоения:

```
document.cookie='ИмяКуки=Значение; expires=дата; path=путь; domain=домен;
secure';
```

то прежние хранившиеся куки не будут стерты, как можно заподозрить. Вместо этого браузер проверит, не имеется ли уже в `document.cookie` куки с именем `ИмяКуки`. Если

нет, то новая куки будет **добавлена** в document.cookie; если да, то для куки с этим именем будут **обновлены** указанные в команде параметры (значение, срок действия и т.д.). Это поведение куки демонстрирует следующий пример:

```
   document.write(document.cookie+'<BR>');
document.cookie = 'basket=SiemensA35';
   document.write(document.cookie+'<BR>');
document.cookie = 'customer=Ivanov';
   document.write(document.cookie+'<BR>');
document.cookie = 'basket=Nokia3310';
   document.write(document.cookie+'<BR>');
```

Напишем универсальную функцию для задания куки, которой Вы можете пользоваться на практике. Первые два ее аргумента (name и value) обязательны, остальные необязательны. В ней используется функция escape(), которая преобразует специальные символы в их коды, например, пробел в %20, равенство в %3D и т.д.

```
function setCookie(name, value, exp, pth, dmn, sec)
{ // Создает cookie с указанными параметрами
 document.cookie = name + '=' + escape(value)
 + ((exp)? '; expires=' + exp : '')
 + ((pth)? '; path='    + pth : '')
 + ((dmn)? '; domain='  + dmn : '')
 + ((sec)? '; secure'         : ''); }
```

Остается научиться передавать этой функции время истечения срока действия куки в правильном формате (пример см. выше). В этом нам поможет метод toGMTString() объекта Date. На практике заранее известно не время истечения срока действия куки, а сам срок действия (в днях, часах, минутах и т.д.), отсчитывая от текущего момента. Напишем функцию, которая по этим данным возвращает точный момент времени, причем в нужном нам формате:

```
function TimeAfter(d,h,m)
{ // Выдает время через d дней h часов m минут
 var now = new Date(),    // объект класса Data
   nowMS = now.getTime(), // в миллисекундах (мс)
   newMS = ((d*24 + h)*60 + m)*60*1000 + nowMS;
   now.setTime(newMS);    // новое время в мс
   return now.toGMTString(); }
```

8.1.4 Удаление cookie

Удалить куки — значит в качестве времени истечения куки указать какой-либо прошлый момент времени, например, "сутки назад". Напишем соответствующую функцию:

```
function deleteCookie(CookieName)
{ // Удаляет куки с данным именем
 setCookie(CookieName,'',TimeAfter(-1,0,0)); }
```

8.1.5 Демонстрационный пример

Теперь у нас есть весь арсенал функций работы с куки; сохраните их единый в файл cookies.js. Мы создадим две **независимые web-**страницы, которые благодаря куки

смогут обменяться информацией. Первая страница запрашивает имя пользователя и записывает его в куки сроком на **1** минуту.

```
<SCRIPT SRC='cookies.js'></SCRIPT>
<SCRIPT>
setCookie('customername',
          prompt('Введите ваше имя',''),
          TimeAfter(0,0,1)  );
alert('Мы Вас запомнили!');
</SCRIPT>
```

Пример 8.1. Запись имени пользователя в cookie

Вторая страница приветствует пользователя по имени, которое она прочитает из **cookie**. Убедитесь, что она узнает пользователя, даже если перед открытием страницы закрыть браузер. Однако если открыть эту страницу через **1** минуту, то она уже не сможет узнать пользователя.

```
<SCRIPT SRC='cookies.js'></SCRIPT>
<SCRIPT>
if(existsCookie('customername'))
     alert('Приветствуем Вас, ' +CookieValue('customername')+ '!');
else alert('Извините, мы Вас уже не помним...')
</SCRIPT>
```

Пример 8.2. Чтение имени пользователя из cookie

8.1.6 Управление фокусом

Фокус — это характеристика текущего окна, фрейма или поля формы. В каждом из разделов, описывающем программирование этих объектов, мы, так или иначе, касаемся вопроса фокуса. Под фокусом понимают возможность активизации свойств и методов объекта. Например, окно в фокусе, если оно является текущим, т.е. лежит поверх всех других окон и исполняются его методы или можно получить доступ к его свойствам.

В данном разделе мы рассмотрим управление фокусом в

- окнах;
- фреймах;
- полях формы.

Следует сразу заметить, что фреймы — это тоже объекты класса `Window`, и многие решения, разработанные для окон, справедливы и для фреймов.

Управляем фокусом в окнах

Для управления фокусом у *объекта* класса "окно" существует два метода: `focus()` и `blur()`. Первый передает фокус в окно, в то время как второй фокус из окна убирает. Рассмотрим простой пример:

```
<SCRIPT>
function open_hidden_window()
{
wid=window.open('','test', 'width=100,height=100');
wid.opener.focus();
wid.document.open();
wid.document.write('<H1>Скрытое</H1>');
wid.document.close();
```

```
}
</SCRIPT>
<INPUT TYPE=button value='Скрытое окно'
   onClick='open_hidden_window()'>
```

В данном примере новое окно открывается и сразу теряет фокус — прячется за окном-родителем. При первом нажатии на кнопку оно еще всплывает и только после этого прячется, но при повторном нажатии пользователь не видит появления нового окна, т.к. оно уже открыто и меняется только его содержимое.

Для того чтобы этого не происходило, нужно после открытия передавать фокус новому окну:

```
<SCRIPT>
function open_visible_window()
{
wid=window.open('','test', 'width=100,height=100');
wid.focus();
wid.document.open();
wid.document.write('<H1>Видимое</H1>');
wid.document.close();
}
</SCRIPT>
<INPUT TYPE=button value='Видимое окно'
   onClick='open_visible_window()'>
```

Если теперь совместить эти два примера на одной странице и нажимать попеременно кнопки "Скрытое окно" и "Видимое окно", то окно будет то появляться, то исчезать. При этом новых окон не появляется, так как с одним и тем же именем может быть открыто только одно окно.

Невидимое окно может доставить пользователю неприятности, из которых самая безобидная — отсутствие реакции на его действия. Код просто записывается в невидимое окно. Но ведь в скрытом окне можно что-нибудь и запустить. Для этого стоит только проверить, существует ли данное окно или нет, и если оно есть и не в фокусе, то активизировать в нем какую-нибудь программу.

Для реализации такого сценария достаточно использовать метод окна `onblur()`. Мы воспользуемся этим методом "в лоб":

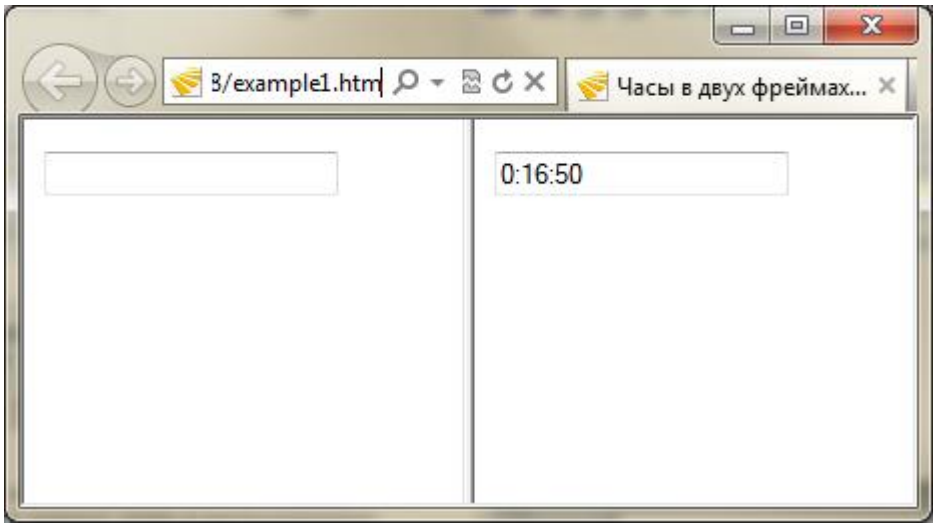
```
window.onblur  = new
 Function("window.defaultStatus = 'Работаем в фоне...';");
window.onfocus = new
 Function("window.defaultStatus = 'Готово';");
```

Этот пример демонстрирует возможность выполнения функции в фоновом режиме. Для проверки этого примера откройте его в браузере, а затем откройте любое другое окно небольшого размера (например, блокнот) и следите за полем статуса в открытом окне браузера. Аналогичного эффекта можно было достичь, поместив указанные команды в контейнере `BODY` в обработчиках событий `onBlur` и `onFocus`.

Конечно, когда разработчики создавали всю эту конструкцию, думали не о том, как насолить пользователю, а о том, как сократить ресурсы, необходимые браузеру для отображения нескольких окон. Ведь можно выполнить все то же самое с точностью до наоборот: запускать, например, часы в фокусе и останавливать их в фоне. Но этот пример мы рассмотрим в контексте фреймов.

Управление фокусом во фреймах

Фрейм — это такое же окно, как и само окно браузера. Точнее, это *объект* того же класса. К нему применимы те же методы, что и к обычному объекту "окно".



Рассмотрим страницу из двух одинаковых фреймов:

```
<HTML>
<FRAMESET COLS='50%,*'>
<FRAME SRC=clock.htm>
<FRAME SRC=clock.htm>
</FRAMESET>
</HTML>
```

Пример 8.3. Часы в двух фреймах (работают там, где фокус)

В файл `clock.htm` поместим следующую страницу, которая тем самым будет отображаться в обоих фреймах:

```
<HTML><HEAD><SCRIPT>
var flag=false;
function clock()
{ if(flag)
  {
   var d = new Date();
   document.f.e.value =
   d.getHours()   + ':' +
   d.getMinutes() + ':' +
   d.getSeconds();
  }
setTimeout('clock();',100);
}
</SCRIPT></HEAD>

<BODY onLoad='clock()'
onFocus='this.flag=true'
onBlur='this.flag=false'>

<FORM NAME=f>
<INPUT NAME=e>
</FORM></BODY></HTML>
```

Пример 8.4. Часы запускаются, если данное окно в фокусе

Если кликнуть на любом из двух фреймов, то пойдут часы именно в этом фрейме, а в другом фрейме, если они уже были запущены ранее, остановятся.

Фокус в полях формы

Обработчики событий `onFocus` и `onBlur` определены для любого поля формы (текстовое однострочное поле, текстовая область, кнопка, радиокнопки, ниспадающие списки). В качестве одного из возможных применений этих обработчиков рассмотрим защиту поля от ввода (изменения) содержимого:

```
<INPUT TYPE=text onFocus='this.blur();' VALUE='Попробуйте изменить'>
```

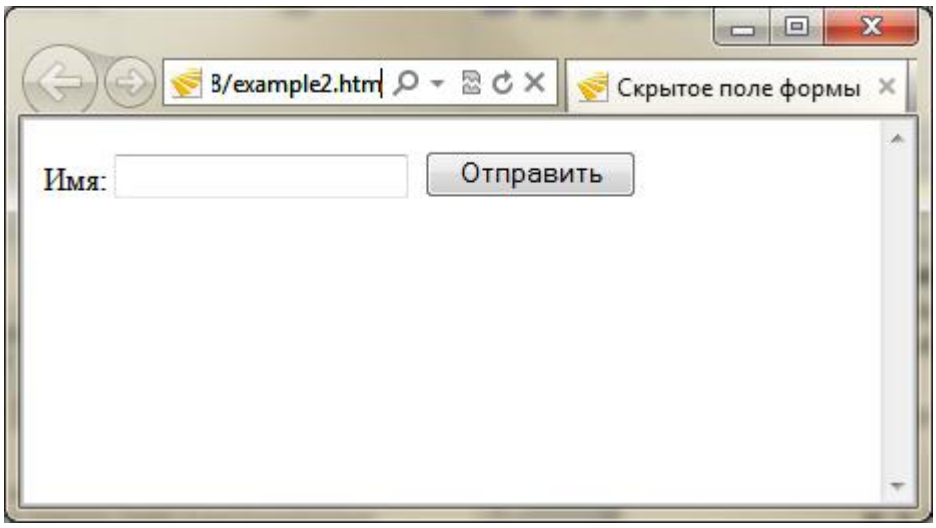
Когда пользователь попытается установить курсор в таком поле ввода, тем самым передав ему фокус, наш обработчик события уведет из него фокус и тем самым изменение поля становится невозможно.

Примечание. Этот пример приведен лишь в качестве демонстрации обработчиков событий. В настоящее время для защиты поля от изменения достаточно указать у него атрибут `READONLY`, т.е. предыдущий пример равносилен следующему:

```
<INPUT TYPE=text READONLY VALUE='Попробуйте изменить'>
```

8.1.7 Скрытая передача данных из форм

Обмен данными в **Web**-технологии подробно рассматривается в **[1]**. Здесь же мы рассмотрим возможности передачи скрытых от пользователя данных с использованием **JavaScript**. Начнем с простого примера.



Пример **8.2**. Введите в поле имя и нажмите кнопку:

```
<FORM NAME=f>Имя:
 <INPUT TYPE=text NAME=user>
 <INPUT TYPE=hidden NAME=h>
 <INPUT TYPE=submit VALUE="Отправить">
</FORM>
<SCRIPT>
 document.f.h.value=window.navigator.appName;
</SCRIPT>
```

Пример 8.5. Отправка данных из скрытых полей формы

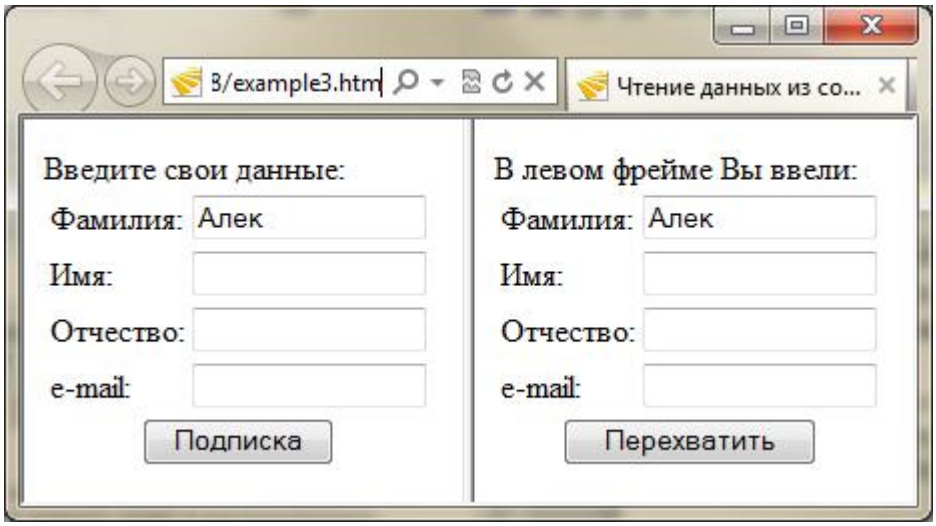
После нажатия кнопки в адресной строке вы увидите, что помимо `user=имя` имеется также `h=имя_вашего_браузера`. В заполненной Вами форме поля `h` не было видно. Таким образом, форма передала на сервер дополнительную информацию помимо Вашего желания. Это уже неприятно, хотя сама информация в данном случае (имя браузера) не представляет из себя ничего криминального. Если бы в качестве метода передачи данных был использован не `GET` (как в нашем примере, по умолчанию), а `POST`, то этой скрытой передачи данных пользователь даже не заметил бы.

Пример **8.3.** Пример состоит из двух фреймов (расположенных в файлах `left.htm` и `right.htm`), в которые помещена одинаковая форма. В правый фрейм, помимо этого, помещен также следующий скрипт:

```
function copyFields()
{ here = document.forms[0].elements;
 there = window.top.frames[0].document.forms[0].elements;

 here[0].value = there[0].value;
 here[1].value = there[1].value;
 here[2].value = there[2].value;
 here[3].value = there[3].value;

setTimeout('copyFields()',100);
}
window.onload=copyFields;
```



Функция `copyFields()` запускается раз в **0,1** сек. Когда пользователь вводит данные в левом фрейме, эти же данные попадают в соответствующие поля правого фрейма. Таким образом, данные из одного окна могут быть считаны программой из другого окна (или фрейма). Вопрос только в том, хотите ли вы, чтобы это происходило. Как решаются эти вопросы, рассказано ниже в разделе "Модель безопасности".

Еще один пример — отправка данных по событию без наличия видимой формы на **web**-странице:

```
<FORM NAME=f METHOD=post ACTION="javascript:alert('Передали данные');">
 <INPUT NAME=h TYPE=hidden>
</FORM>
<SCRIPT>
 document.f.h.value = window.navigator.appName;
</SCRIPT>
<A HREF="javascript:alert('Просто ссылка');"
onClick="document.f.submit();">Нажми на ссылку</A>
```

При нажатии на гипертекстовую ссылку произойдет не только выдача сообщения, которое в этой ссылке указано, но и отправка данных формы. В итоге вы получите два окна предупреждения. Но второе окно вы не заказывали!

Конечно, бесконтрольной передачи данных на сервер можно избежать, введя режим подтверждения отправки (в настройках браузера). Но, во-первых, многие пользователи его отключают, а во-вторых, можно использовать не формы, а, например, графику. И эту возможность мы сейчас и рассмотрим в следующем разделе.

8.1.8 Невидимый код

Вопрос доступности **JavaScript**-кода рассматривается с двух точек зрения: идентификация, как следствие — необходимость сокрытия кода, и безопасность пользователя, следовательно — доступность кода.

Приемы программирования со скрытым кодом позволяют решить еще несколько задач, которые не связаны с безопасностью.

Мы будем рассматривать возможности использования скрытого кода, не вынося вердиктов о преимуществе того или иного подхода. Рассмотрим несколько вариантов:

- невидимый фрейм;
- код во внешнем файле;
- обмен данными посредством встроенной графики.

Строго говоря, первые два варианта не скрывают код полностью. Они рассчитаны либо на неопытных пользователей, либо на нелюбопытных. Так или иначе, не каждый же раз вы будете смотреть исходный текст страницы.

Невидимый фрейм

Технология программирования в невидимом фрейме основана на том, что при описании фреймовой структуры можно задать конфигурацию типа:

```
<FRAMESET COLS="100%,*">
   <FRAME SRC=left.htm>
   <FRAME SRC=right.htm>
</FRAMESET>
```

Пример 8.6. Правый фрейм имеет нулевую ширину (граница видима)

В этом случае левый фрейм займет весь объем рабочей области окна, а содержание правого будет скрыто. Именно в этом невидимом фрейме мы и разместим код программы (например, приведенный выше скрипт считывания полей из формы в левом фрейме). В невидимый фрейм иногда помещают функции подкачки графики, позволяя пользователю уже работать с основным фреймом, пока грузится остальная часть графики.

В данном примере, однако, мы оставили пользователю возможность разоблачить нас: вдоль правой границы окна видна вертикальная полоска — это граница между фреймами. Пользователь может подвинуть ее влево и увидеть правый фрейм. Защититься от этого несложно — достаточно задать толщину границ фреймов, а также указав невозможность изменять размеры у каждого фрейма:

```
<FRAMESET COLS="100%,*" BORDER=0>
   <FRAME NORESIZE SRC=left.htm>
```

```
   <FRAME NORESIZE SRC=right.htm>
</FRAMESET>
```

Пример 8.7. Правый фрейм имеет нулевую ширину (граница невидима)

Код во внешнем файле

О том, как подключать код **JavaScript**, размещенный во внешнем файле, рассказывалось во вводной лекции:

```
<SCRIPT SRC="myscript.js"></SCRIPT>
```

Данный способ позволяет скрыть код лишь от ленивого пользователя. Но сам код **JavaScript** легко доступен, т.к. указанный файл можно просто скачать отдельно, либо сохранить всю **HTML-**страницу **(**со всеми подключенными к ней скриптами**)** с помощью меню браузера.

Обмен данными посредством встроенной графики

Данный прием основан на двух идеях: возможности подкачки графического образа без перезагрузки страницы и возможности подкачки этого графического образа не через указание **URL** графического файла, а через **CGI-**скрипт, который возвращает `Content-type: image/...` или осуществляет перенаправление. При этом следует учитывать, что использовать метод, отличный от **GET**, можно только в формах. В следующем примере мы создали функцию `change_image()`, которая формально говоря меняет значение свойства `src` картинки. Но в качестве побочного эффекта позволяет серверу узнать, установлены ли у пользователя **cookie (**если соответствующим образом запрограммировать **CGI-**скрипт `image.cgi` на стороне сервера**)**:

```
<SCRIPT>
function change_image(x)
{
 document.x.src = 'http://abc.ru/image.cgi?' + document.cookie;
}
</SCRIPT>

<A HREF="javascript:change_image(i);"><IMG NAME=i SRC=image1.gif></A>
```

Эта безобидная последовательность операторов **JavaScript** позволит нам узнать получил ли клиент **cookie.** Куки могут не поддерживаться по разным причинам. В данном случае программа передает на сервер выставленные им **cookie** в качестве параметра скрипта под видом изменения картинки.

8.1.9 Модель безопасности

При программировании на **JavaScript** потенциально существует возможность доступа из программы к персональной информации пользователя. Такая проблема возникает всегда, когда нечто, запускаемое на компьютере, имеет возможность самостоятельно организовать обмен данными по сети с удаленным сервером. От версии к версии управление защитой таких данных постоянно совершенствуется, но всегда нужно иметь в виду, что множество "следопытов" исследует эту проблему и постоянно открывает все новые и новые возможности обхода механизмов защиты. Объясним только основные моменты в принципах защиты информации в **JavaScript**, а поиск потенциально слабых мест оставим в качестве домашнего задания для наиболее пытливых читателей.

По умолчанию к защищенным в **JavaScript** данным относятся:

Таблица 8.1. Свойства и методы, являющиеся защищенными

Объект или класс	Свойства
document	cookie, domain, lastModified, location, referrer, title, URL, links[], forms[]
Form	action
элемент формы	checked, defaultChecked, name, value, defaultValue, selectedIndex, toString()
history	current, next, previous, toString(), все элементы массива посещенных адресов
Location, Link, Area	hash, host, hostname, href, pathname, port, protocol, search, toString()
Option	defaultSelected, selected, text, value
Window	defaultStatus, status

Защищенными эти данные являются с той точки зрения, что программа не может получить значения соответствующих атрибутов. Главным образом речь здесь идет о программе, которая пытается получить доступ к данным, которые определены на другой странице (не на той, в рамках которой данная программа исполняется). Например, к данным из другого окна.

В настоящее время известны три модели защиты: **запрет на доступ (Navigator 2.0)**, **taint model (Navigator 3.0)**, **защита через Java (Navigator 4.0)**. Применение моделей и соответствующие приемы программирования — это отдельный сложный вопрос, требующий знаний и навыков программирования на языке **Java**, поэтому в рамках данного курса мы его рассматривать не будем.

Отметим только, что к большинству свойств объектов текущего окна программист имеет доступ. Они становятся защищенными только в том случае, если относятся к документу в другом окне и загруженному из другого **Web-**узла. Поэтому ограничения, накладываемые системой безопасности **JavaScript**, достаточно гибкие и не очень сильно мешают разработке страниц с применением этого языка программирования.

9 ДОП. ЛЕКЦИЯ: ФОРМАТ И СИНТАКСИС COOKIE

9.1.1 Спецификация

Полное описание поля `Set-Cookie` **HTTP-**заголовка**:**

```
Set-Cookie: NAME=VALUE; expires=DATE;
    path=PATH; domain=DOMAIN_NAME;
        secure
```

Минимальное описание поля `Set-Cookie` **HTTP-**заголовка**:**

```
Set-Cookie: NAME=VALUE;
```

`NAME=VALUE` **-** строка символов**,** исключая перевод строки**,** запятые и пробелы**.** `NAME` **-** имя **cookie,** `VALUE` **-** значение**.**

`expires=DATE` **-** время хранения **cookie,** т**.**е**.** вместо `DATE` должна стоять дата в формате `Wdy, DD-Mon-YYYY HH:MM:SS GMT`**,** после которой истекает время хранения **cookie.** Если этот атрибут не указан**,** то **cookie** хранится в течение одного сеанса**,** до закрытия браузера**.**

`domain=DOMAIN_NAME` **-** домен**,** для которого значение **cookie** действительно**.** Например**,** `domain=cit-forum.com`**.** В этом случае значение **cookie** будет действительно и для сервера `cit-forum.com`**,** и для `www.cit-forum.com`**.** Но не радуйтесь**,** указания двух последних периодов доменных имен хватает только для доменов иерархии **"COM", "EDU", "NET", "ORG", "GOV", "MIL",** и **"INT".** Для доменов иерархии **"RU"** придется указывать три периода**.**

Если этот атрибут опущен**,** то по умолчанию используется доменное имя сервера**,** с которого было выставлено значение **cookie.**

`path=PATH` **-** этот атрибут устанавливает подмножество документов**,** для которых действительно значение **cookie.** Например**,** указание `path=/win` приведет к тому**,** что значение **cookie** будет действительно для множества документов в директории `/win/`**,** в директории `/wings/` и файлов в текущей директории с именами типа `wind.html` и `windows.shtml`

Если этот атрибут не указан**,** то значение **cookie** распространяется только на документы в той же директории**,** что и документ**,** в котором было установлено **cookie.**

`secure` **-** если стоит такой маркер**,** то информация **cookie** пересылается только через **HTTPS (HTTP** с использованием **SSL (Secure Sockets Layer,** протокол защищенных сокетов**)).** Если этот маркер не указан**,** то информация пересылается обычным способом**.**

9.1.2 Синтаксис HTTP заголовка для поля Cookie

Когда запрашивается документ с **HTTP** сервера**,** браузер проверяет свои **cookie** на предмет соответствия домену сервера и прочей информации**.** В случае**,** если найдены удовлетворяющие всем условиям значения **cookie** браузер посылает их серверу в виде пары `имя/значение`**:**

```
Cookie: NAME1=OPAQUE_STRING1;
    NAME2=OPAQUE_STRING2
...
```

9.1.3 Дополнительные сведения

В случае, если **cookie** принимает новое значение при имеющемся уже в браузере **cookie** с совпадающими `NAME`, `domain` и `path`, старое значение затирается новым. В остальных случаях новые **cookies** добавляются.

Использование **expires** не гарантирует сохранность **cookie** в течение заданного периода времени, поскольку клиент (браузер) может удалить запись вследствие нехватки выделенного места или каких-либо других лимитов.

Клиент (браузер) имеет следующие ограничения:

- всего может храниться до **300** значений **cookies**;
- каждый **cookie** не может превышать **4** Кбайт;
- с одного сервера или домена может храниться до **20** значений **cookie**.

Если ограничение **300** или **20** превышается, то удаляется первая по времени запись. При превышении **4**К - корректность такого **cookie** страдает - отрезается кусок записи (с начала этой записи) равный превышению.

В случае кэширования документов, например, **proxy-**сервером, поле `Set-cookie` **HTTP-**заголовка никогда не кэшируется.

Если **proxy-**сервер принимает ответ, содержащий поле `Set-cookie` в заголовке, предполагается, что поле таки доходит до клиента вне зависимости от статуса `304` **(Not Modified)** или `200` **(OK)**.

Соответственно, если клиентский запрос содержит в заголовке **Cookie**, то он должен дойти до сервера, даже если установлен `If-modified-since`.

Я полагаю, что все, что сказано про **proxy**, не относится к случаю, когда **cookie** устанавливается жестко с помощью **META-**тегов.

9.1.4 Примеры

Ниже приведено несколько примеров, иллюстрирующих использование **cookies**

Первый пример:

Клиент запрашивает документ и принимает ответ:

```
Set-Cookie: CUSTOMER=WILE_E_COYOTE;
    path=/; expires=Wednesday,
        09-Nov-99 23:12:40 GMT
```

Когда клиент запрашивает **URL** с путем `"/"` на этом сервере, он посылает:

```
Cookie: CUSTOMER=WILE_E_COYOTE
```

Клиент запрашивает документ и принимает ответ:

```
Set-Cookie:
    PART_NUMBER=ROCKET_LAUNCHER_0001;
        path=/
```

Когда клиент запрашивает **URL** с путем `"/"` на этом сервере, он посылает:

```
Cookie: CUSTOMER=WILE_E_COYOTE;
PART_NUMBER=ROCKET_LAUNCHER_0001
```

Клиент получает:

```
Set-Cookie: SHIPPING=FEDEX; path=/foo
```

Когда клиент запрашивает **URL** с путем `"/"` на этом сервере, он посылает:

```
Cookie: CUSTOMER=WILE_E_COYOTE;
PART_NUMBER=ROCKET_LAUNCHER_0001
```

Когда клиент запрашивает **URL** с путем `"/foo"` на этом сервере, он посылает:

```
Cookie: CUSTOMER=WILE_E_COYOTE;
    PART_NUMBER=ROCKET_LAUNCHER_0001;
    SHIPPING=FEDEX
```

Второй пример:

Клиент принимает:

```
Set-Cookie:
    PART_NUMBER=ROCKET_LAUNCHER_0001;
        path=/
```

Когда клиент запрашивает **URL** с путем `"/"` на этом сервере, он посылает:

```
Cookie: PART_NUMBER=ROCKET_LAUNCHER_0001
```

Клиент принимает:

```
Set-Cookie:
    PART_NUMBER=RIDING_ROCKET_0023;
        path=/ammo
```

Когда клиент запрашивает **URL** с путем `"/ammo"` на этом сервере, он посылает:

```
Cookie: PART_NUMBER=RIDING_ROCKET_0023;
    PART_NUMBER=ROCKET_LAUNCHER_0001
```

Комментарий: здесь мы имеем две пары `имя/значение` с именем `"PART_NUMBER"`.

Это наследие из предыдущего примера, где значение для пути `"/"` прибавилось к значению для `"/ammo"`.

1. Курс "Введение в CGI". http://www.intuit.ru/department/internet/cgi/