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).

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.

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 (html, txt)

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 $(...((N)^2)^2...)^2 = N^{2K}$.

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`.

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.

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)`

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.

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:

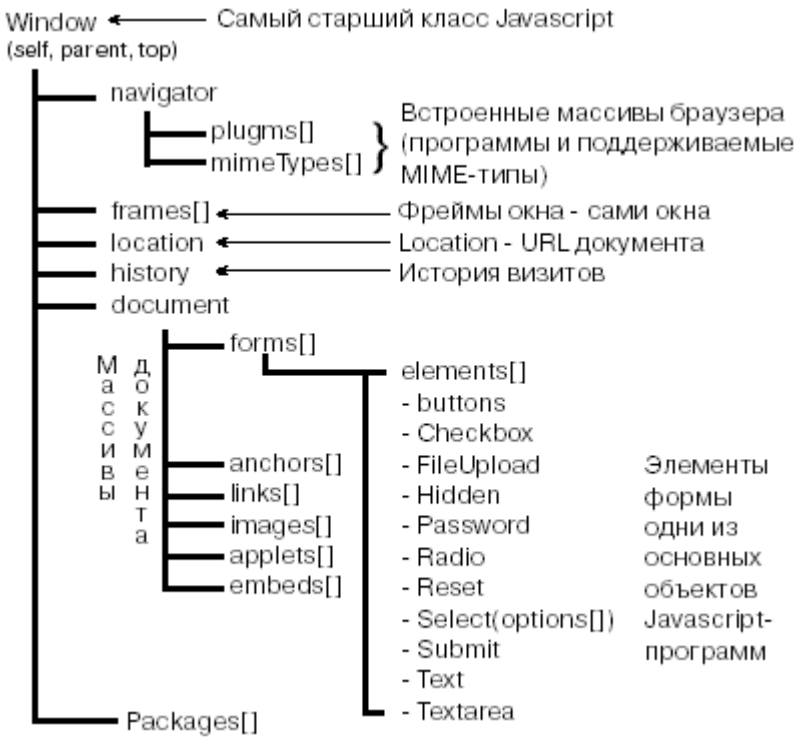

Рис. 3.1. Иерархия объектов DOM (фрагмент)

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 ра**

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)
`function_name.arguments[]`	Все аргументы, переданные функции `function_name()` при вызове

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://intuit.ru/">intuit</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.

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://intuit.ru/">intuit</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://intuit.ru/">intuit</A>
<SCRIPT> document.links[0].click(); </SCRIPT>
```

При открытии такой страницы пользователь сразу будет перенаправлен на сайт ИНТУИТ. 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.

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 (html, txt)

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.

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.

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.

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 (html, txt)

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.

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