18. Программирование «за кадром»

Цель: ознакомиться с нюансами программирования «за кадром», научиться применять такое программирование на практике.

18.1. Теоретические сведения

Обычно описание языка программирования начинают с *типов* и структур данных, операторов и *функций,* а заканчивают библиотеками стандартных *функций* (главным образом, ввода/вывода) и опциями компилятора. В своем описании JavaScript мы двигались в обратном порядке и рассказываем об этом в конце нашего курса.

18.1.1. Типы и структуры данных

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

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

При этом все они делятся на встроенные и определяемые программистом.

18.1.2. 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:

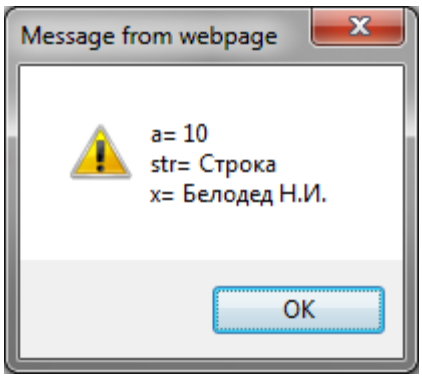
```
числовой литерал: 10
числовой литерал: 2.310
числовой литерал: 2.3e+2
строковый литерал: 'Это строковый литерал'
строковый литерал: "Это строковый литерал"
```

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

```
<script language="javascript" type="text/javascript">
   var a = 10;
   var str = 'Строка';
   var x = "Белодед Н.И.";

   if (x == 'test')
      window.alert(x);
   else {
      window.alert("a= " + a + '\n' + "str= " + str + '\n' + "x= " + x);
   }

</script>
```



Два варианта строковых литералов необходимы для того, чтобы использовать вложенные строковые литералы. Вообще говоря, есть подозрение, что равноправие ."..." и '...' мнимое.

Это связано с особенностями реализации Netscape. Дело в том, что прямое переназначение неправильно отображает кириллицу в **Win32,** а вот косвенное работает. Похоже, что "..." разрешает анализ информации внутри строкового литерала JavaScript-интерпретатором, а '...' — нет.

Если быть более точным, то следует сказать, что строка - это *объект.* У этого *объекта* существует великое множество методов. Строчный литерал и строчный *объект* - далеко не одно и то же. При

применении к строчным литералам методов строчных *объектов* происходит преобразование первых в последние.

18.1.3. Переменные

Переменные в JavaScript могут быть определены назначением или при помощи оператора var:

```
<script language="javascript" type="text/javascript">
   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));

</script>
```

Как видно из примера, переменные могут принимать самые разные значения, при этом тип переменной определяется контекстом.

Переменная является свойством окна. Например, мы можем открыть окно, определить в нем новую переменную и использовать ее.

Существуют ли в JavaScript различные типы переменных? По всей видимости, да. При объявлении переменной тип не указывается. Тип значения определяется контекстом, поэтому можно было бы предположить, что все переменные — одного и того же типа. Однако очевидно, что присваивание переменной значения *объекта* окна (window. open()) или *объекта* потока (setTimeoutO), вызывает создание в памяти совершенно разных структур.

Также JavaScript поддерживает полиморфизм, т.е. существует два разных *объекта* с одинаковыми именами, и система в них не путается.

18.1.4. Массивы

Массивы делятся на встроенные (document.links[], document. images[],...) и определяемые пользователем (автором документа). Для массивов задано несколько методов:

- join();
- reverse();
- sort().

И свойство length, которое позволяет получить число элементов массива. Это свойство активно используется в примерах данного раздела. В частности, при обсуждении метода join().

Для определения массива пользователя существует специальный конструктор:

```
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
```

Как видно, массив может состоять из разнородных элементов. Массивы не могут быть многомерными.

Для работы с массивами в JavaScript применяются методы join(), reverse(), sort(). Кроме того, массивы обладают свойством длины, length.

18.1.4.1. Метод join()

Метод join() позволяет объединить элементы массива в одну строку. Он является обратной *функцией* методу split(), который применяется к *объектам* типа STRING.

Пример использования метода join() — замена символа в строке:

```
var str = "document.img1.src='http://images/image1.gif';"
```

Исходная строка:

str= **document.img1.src='http://images/image1.gif';**

Заменяем в строке все единицы на двойки:

```
var b = str.split('1');
str = b.join('2');
```

Получаем следующий результат:

str= **document.img2.src='http://images/image2.gif';**

Последний пример показывает, что массив пользователя можно получить и без явного применения конструктора массива. Массив элементов строки получается просто как результат действия *функции* split().

18.1.4.2. Метод reverse()

Метод reverse() применяется для изменения на противоположный порядка элементов массива внутри массива. Предположим, массив натуральных чисел упорядочен по возрастанию:

```
var a = new Array(1, 2, 3, 4, 5);
```

Упорядочим его по убыванию:

```
a.reverse();
```

18.1.4.3. Метод sort()

Как принято в современных интерпретируемых языках, например в Perl, метод sort() позволяет отсортировать элементы массива в соответствии с некоторой *функцией* сортировки, чье имя используется в качестве аргумента метода:

```
var a = new Array(1, 6, 9, 9, 3, 5);

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

b = a.sort(g);
```

18.1.5. 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 *объект* 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 — тип данных;
- функция — объект;
- конструкторы объектов.

18.1.5.1. Функция - тип данных

Определяют *функцию* при помощи ключевого слова function:

```
function f_name(arg1,arg2,...)
{
        /* function body */
}
```

Здесь следует обратить внимание на следующие моменты. Во-первых, function определяет переменную f_name. Эта переменная имеет тип «function».

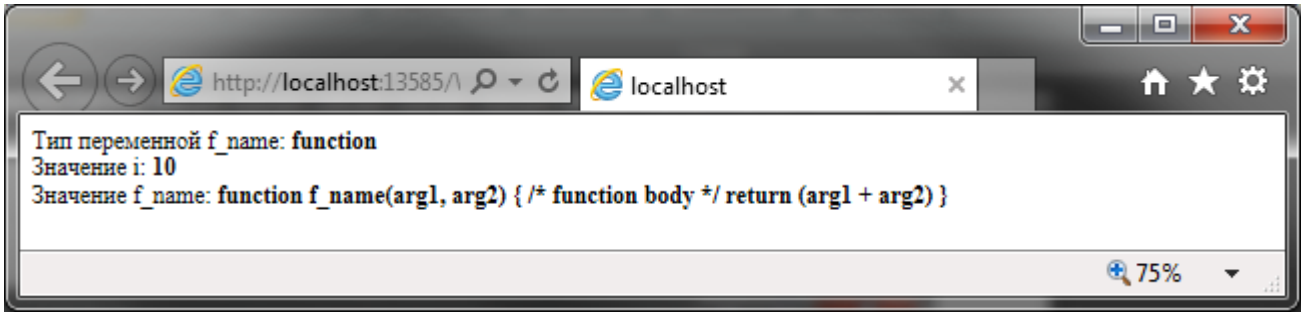
Во-вторых, этой переменной присваивается значение.

```
<script language="javascript" type="text/javascript">
function f_name(arg1, arg2)
{
   /* function body */
   return (arg1 + arg2)
}
var i = f_name(3, 7);

document.write("Тип переменной f_name: <b>" + typeof (f_name) + "</b><br>");
```

```
document.write("Значение i: <b>" + i.valueOf() + "</b><br>");
document.write("Значение f_name: <b>" + f_name.valueOf() + "</b><br>");

</script>
```



Очевидно, что если *функцию* можно присвоить переменной, то ее можно передать и в качестве аргумента другой *функции.* Все это усиливается при использовании *функции* eval(), которая позволяет реализовать отложенное исполнение JavaScript-кода.

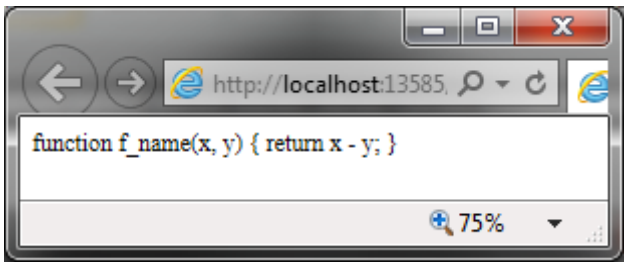
18.1.5.2. Функция – объект

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 — объект* Function.

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

```
<script language="javascript" type="text/javascript">
   function f_name(x, y) {
      return x - y;
   }
   document.write(f_name.toString() + "<br>");

</script>
```



Свойства *функции* доступны для программиста только тогда, когда они вызываются внутри *функции.* При этом обычно программисты имеют дело с массивом аргументов *функции* (arguments[]), его длиной (length), именем *функции,* вызвавшей данную *функцию* (caller), и *прототипом* (prototype).

Если *функция* может быть вызвана из других *функций,* то в этом случае используется свойство caller.

Еще одним свойством объекта Function является prototype, но это общее свойство всех *объектов,* поэтому и обсуждать его мы будем в контексте *типа данных* Object. Упомянем только о конструкторе *объекта F*unction:

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

Здесь f — это объект класса Function. Его можно использовать и как обычную *функцию.* Конструктор используют для получения безымянных *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.

18.1.6. Объекты

Объект - это главный *тип данных* JavaScript. Любой другой *тип данных* имеет объектовую «обертку» — Wrapper. Ото 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 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.

18.1.6.1. Понятие объекта

Сначала рассмотрим пример произвольного, определенного пользователем *объекта,* потом выясним, что же это такое:

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 классов *объектов* пользователя. Гораздо проще писать *функции* для каждой страницы.

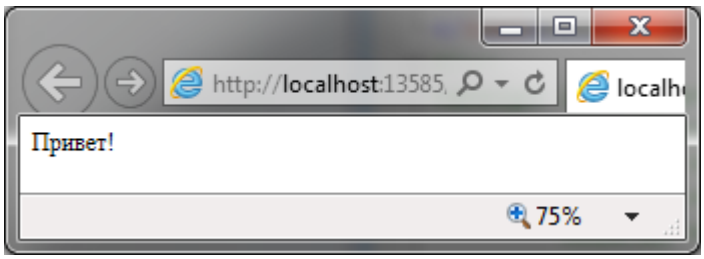
18.1.6.2. Прототип

Обычно мы имеем дело со встроенными *объектами* JavaScript. Интересно свойство *объектов,* которое носит название prototype. Прототип - это другое название конструктора *объекта* конкретного класса. Например, если мы хотим добавить метод к *объекту* класса String:

```
<script language="javascript" type="text/javascript">
   String.prototype.out = new Function("a", "a.write(this)");
   "Привет!".out(document);

</script>
```

Результат исполнения:



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 можно добавить метод к встроенному в JavaScript классу *объектов.* Основная идея заключается в том, чтобы переопределить конструктор раньше, чем он будет использован. HTML-парсер разбирает HTML и создает встроенные *объекты* раньше, чем запускается JavaScript-интерпретатор. Из этого следует, что *объект* на странице нужно создать через JavaScript-код. В этом случае сначала происходит переопределение *объекта* Image, а уже после этого создается встроенный *объект* данного класса. При работе с Internet Explorer все иначе. Если на свойство prototype у строкового *объекта* он не «ругается», то для Image такое свойство уже не определено.

18.1.6.3. Методы объекта Object

Object — это *объект* и, следовательно, у него могут быть методы. Таких методов мы рассмотрим три: toString(), valueOf () и assign().

Метод toString() осуществляет преобразование *объекта* в строку символов. Он используется в JavaScript-программах повсеместно, но неявно. Например, при выводе числа или строковых *объектов.* Интересно применение toString() к *функциям:*

```
document.write("prot.toString()= <b>" + prot.toString() + "</b><br>");
```

Результат исполнения:

function prot() { a = this.src.split(':') protocol = a[0] + ":"; return protocol; }

Здесь мы используем *функцию* prot() из примера с прототипом. Если распечатать таким же образом *объект* Image, то получим следующее:

картинка:[object]

Таким образом, далеко не всегда метод toString() возвращает строковый эквивалент содержания *объекта.* Он может просто вернуть его тип. Internet Explorer при этом возвращает «Object», в то время как Netscape Navigator - «object Image».

Аналогично ведет себя и метод valueOf(). Этот метод позволяет получить значение объекта. В большинстве случаев он работает подобно методу toString(), особенно если нужно выводить значение на страницу:

```
document.write("prot.valueOf()= <b>" + prot.valueOf() + "</b><br>");
```

Результат исполнения:

function prot() { a = this.src.split(':') protocol = a[0] + ":"; return protocol; }

Как видим, результат тот же, что и в методе toString().

В отличие от двух предыдущих методов, assign() позволяет не прочитать, а переназначить свойства и методы объекта. Данный метод используется в контексте присваивания объекту некоторого значения:

```
object = value; <=> object.assign(value);
```

18.1.7. 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:

- var;
- {...}
- if;
- while;
- for;
- for ... in;
- break;
- continue;
- return.

Сразу оговоримся, что этот список неполный.

18.1.7.1. var

Оператор var служит для объявления переменной. При этом переменная может принимать значения любого из разрешенных типов данных. На практике довольно часто обходятся без явного использования var. Переменная соответствующего типа создается путем простого присваивания:

```
var a;
var a = 10;
var a = new Array();
var a = new Image();
```

Все перечисленные выше примеры использования var верны и могут быть применены в JavaScript-программе. Область действия переменной определяется блоком (составным оператором), в котором используется переменная. Максимальная область действия переменной — страница.

18.1.7.2. {...}

Фигурные скобки определяют составной оператор JavaScript — блок. Они одновременно ограничивают область действия переменных, которые определены внутри этих скобок. За пределами блока переменные не видны:

```
{
   var i = 0;
}
```

Основное назначение блока - определение тела цикла и тела условного оператора.

18.1.7.3. if

Условный оператор применяется для ветвления программы по некоторому логическому условию. Общий синтаксис:

```
if (логическое выражение) оператор1;
[else оператор2;]
```

Логическое выражение — это выражение, которое принимает значение true или false. Если оно равно true, то оператор 1 исполняется.

18.1.7.4. while

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

```
While (логическое выражение) оператор;
```

Оператор, в том числе и составной, — тело цикла. Тело исполняется до тех пор, пока верно логическое условие:

```
while (flag == 0) {
   id = setTimeout("test();", 500);
}
```

Обычно цикл этого типа применяют при выполнении периодических действий до некоторого события.

18.1.7.5. for

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

```
for (инициализация переменных цикла;
условие; модификация переменных цикла) оператор;
```

Оператор в теле цикла может быть блоком.

18.1.7.6. for... in

Данный оператор позволяет «пробежаться» по свойствам *объекта.* Рассмотрим пример:

```
<script language="javascript" type="text/javascript">
   for (v in window.document) {
      document.write(v + ": <b>" + eval('document.' + v) + "</b><br>");
   }

</script>
```

В результате выводятся все свойства текущего *объекта* «документ».

18.1.7.7. break

Оператор break позволяет досрочно покинуть тело цикла. Распечатаем только title документа:

```
<script language="javascript" type="text/javascript">
for(v in window.document) if(v == "title")
{
   document.write(v + ": <b>" + eval('document.' + v) + "</b><br>");
   break;
}

</script>
```

18.1.7.8. continue

Того же результата, что и при использовании break, можно было бы достичь при помощи оператора continue:

```
<script language="javascript" type="text/javascript">
   for (v in window.document) {
      if (v != "title") continue;
```

```
        document.write(v + ": <b>" + eval('document.' + v) + "</b><br>");
        break;
    }

</script>
```

Этот оператор позволяет пропустить часть тела цикла (от оператора до конца тела) и перейти к новой итерации. Таким образом, мы просто пропускаем все свойства до title и после этого выходим из цикла.

18.1.7.9. return

Оператор return используют для возврата значения из *функции* или обработчика события (см. разделы «Поле статуса», «Обмен данными»).

18.1.8. 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.

18.1.8.1. 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 language="javascript" type="text/javascript">
function hide_window()
{
   wid = window.open("","test","width=400,height=200");
   wid.opener.focus();
   wid.document.open();
   wid.document.close();
}

</script>
```

В данном примере новое окно открывается и сразу теряет фокус; прячется за основным окном-родителем. Если при первичном нажатии на кнопку оно еще всплывает и только после этого прячется, то при повторном нажатии пользователь не видит появления нового окна, так как оно уже открыто и меняется только его содержимое.

Для того чтобы этого не происходило, нужно после открытия передавать фокус на новое окно:

```
<script language="javascript" type="text/javascript">
function visible_window()
{
        wid=window.open("","test","width=400,height=200");
        wid.focus();
        wid.document.open();
        wid.document.close();
}

</script>
```

Если теперь нажимать попеременно кнопки «Скрытое окно» и «Видимое окно», 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(). Его можно также задать в контейнере BODY в качестве обработчика события onBlur, но в этом случае он видим пользователю. Мы воспользуемся этим методом «в лоб»:

```
<script language="javascript" type="text/javascript">
   window.onblur = new Function("window.defaultStatus ='Background started...';");
   window.onfocus = new Function("window.defaultStatus ='Document:Done';");

</script>
```

18.1.8.2. 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»:

HTMLPage.htm

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
   <title></title>
</head>
<script language="javascript" type="text/javascript">

</script>

   <frameset cols="50%, *">
            <frame src="clock.htm">
            <frame src="clock.htm">
   </frameset>

</html>
```

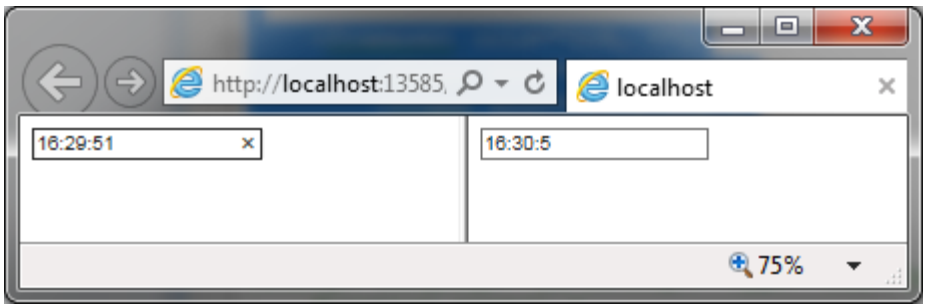
clock.htm

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<script language="javascript" type="text/javascript">
   var flag = 1;
   function clock() {
      if (flag == 0) {
         d = new Date();
         s = d.getHours() + ':' + d.getMinutes() + ':' + d.getSeconds();
         window.document.forms[0].elements[0].value = s;
      }
      setTimeout('clock(); ', 100);
   }
   window.onblur = new Function('this.flag = 1;');
   window.onfocus = new Function('this.flag = 0;');
   window.onload = clock;

</script>
</head>
<body onload='clock()' onfocus='this.flag=1' onblur='this.flag=0'>

<form name="f">
   <input name="e" />
</form>

</body>
</html>
```



Данный фрагмент кода размещен в каждом из двух фреймов, которые отображаются в примере. А их именно два. Просто ширина границы набора фреймов установлена в 0. Если окно примера разделить мысленно пополам и «кликнуть» мышью в одну из половин, то пойдут часы в этой половине. Если теперь переместиться в другой фрейм и «кликнуть» мышью в нем, то часы пойдут в поле формы этого фрейма, а в другом фрейме остановятся.

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". Пример 8.1(кликните, чтобы открыть). Рассмотрим страницу из двух одинаковых фреймов. 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.

HTMLPage.htm

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
   <title></title>
</head>
<script language="javascript" type="text/javascript">

</script>

   <frameset cols="50%, *">
      <frame src="clock.htm">
      <frame src="clock.htm">
   </frameset>

</html>
```

clock.htm

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<script language="javascript" type="text/javascript">
   var flag = false;
   function clock(){
      if (flag == true) {
         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>
```

18.1.9. Фокус в полях формы

Управление фокусом в полях формы, кроме этого раздела, описано еще и в разделе «Текст в полях ввода». Здесь мы рассматриваем этот вопрос в контексте общего применения методов blur() и focus(). Эти методы определены для любого поля формы, а не только для полей ввода. Рассмотрим простой пример.

Попробуйте изменить в этой форме значение любого из полей. Вряд ли это вам удастся. Обработчик события Focus (onFocus) уводит фокус из поля на произвольное место страницы.

18.1.10. Скрытая передача данных из форм

Рассмотрим пример – отправка данных по событию без наличия какой-либо формы в документе.

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
   <title></title>
</head>
<script language="javascript" type="text/javascript">

</script>

<form name="hf" action="javascript: window.alert('Готово'); void(0);" method="post">
   <input name="hfi" type="hidden">
</form>

<script language="javascript" type="text/javascript">
   document.hf.hfi.value = location.href;
</script>

<a href="javascript: window.alert('Внимание'); void(0);" onclick="document.hf.submit(); ">
   Нажми на ссылку
</a>

</html>
```

Согласно примеру при нажатии на гипертекстовую ссылку произойдет не только выдача сообщения, которое в этой ссылке указано, но и событие Submit для 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. Но, во-первых, многие пользователи его отключают, а во-вторых, можно использовать не формы, а, например, графику. И эту возможность мы рассматриваем в разделе «Невидимый код».

18.1.11. 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 возможность использования скрытого кода без выдачи вердиктов о преимуществе того или иного подхода. Рассмотрим несколько вариантов:

- невидимый фрейм;
- код во внешнем файле;
- обмен данными посредством встроенной графики.

Строго говоря, первый и последний варианты 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.

18.1.12. Невидимый фрейм

Технология программирования в невидимом фрейме основана на том, что при описании фреймовой структуры можно задать конфигурацию типа:

```
<frameset cols = "100%, *">
   <frame name=left" src="hcf1.html">
   <frame name="right" src="hcf1.html">
</frameset>
```

При таком размещении страниц по фреймам и фреймов в рабочей области окна левый фрейм займет весь объем рабочей области окна, а содержание правого будет скрыто. Именно в этом невидимом фрейме мы и разместим код программы.

При нажатии на кнопку «Пример невидимого фрейма» откроется новое окно. Если присмотреться внимательно, то кроме картинки с правой стороны окна можно увидеть вертикальную границу. Это граница фрейма. Ее можно двигать. В правый невидимый фрейм мы поместили *функцию* подкачки картинок. Этот

прием позволяет загружать картинки с сервера тогда, когда содержание левого фрейма уже отображено. Если *функцию* разместить в главном окне, то время отображения будет зависеть от многих факторов, например картинки, размещенные в заголовке документа, браузер начнет перекачивать раньше картинок в теле документа.

При последовательном обмене это будет означать увеличение времени загрузки отображаемой части страницы.

18.1.13. Код во внешнем файле

Попав на данную страницу, вы уже использовали программу из внешнего файла. Чтобы убедиться в этом, достаточно посмотреть на HTML-разметку данной страницы:

```
<html>
   <head>
      ...
   <script language="javascript" type="text/javascript" src="../css/jsc.pgm">
   </script>
      ...
   </head>
   <body onload="jump(); ">
      ...
   </body>
</html>
```

Контейнер SCRIPT определяет внешний файл размещения скриптов. *Функция* jump() расположена именно в этом файле. Она анализирует ссылку на данный документ, и если в ней есть компонент hash(#), то она продергивает файл до якоря, указанного в hash. Чтобы в этом убедиться, перейдите по любой внутренней ссылке, например, из меню разбивки раздела на подразделы, а после этого перезагрузите документ по Ctrl+R Сначала документ будет загружен, а потом прокручен до указанного якоря.

18.1.14. 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, и через CGI-скрипт, который возвращает Content-type: image/... или осуществляет перенаправление.

При этом следует учитывать, что использовать метод, отличный от GI I, можно только в формах, а мы хотим просто менять значение свойства src:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
   <title></title>
</head>
<script language="javascript" type="text/javascript">
function change_image(x){
   s = "http://intuit.ru/cgi-bin/image_script?" + document.cookie;
   document.x.src = s;
   ...
}

</script>

<a href="javascript: change_image(i); void(0);">
   <img name="i" src="image1.gif" />
</a>

</html>
```

Эта безобидная последовательность операторов JavaScript позволит нам узнать получил ли клиент cookie. «Волшебные ключики» могут не поддерживаться по разным причинам. В данном случае программа передает на сервер выставленные им «ключики» в качестве параметра скрипта под видом изменения картинки.

18.1.15. 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:

Объект	**Свойства**
Document	cookie, domain, forms[], lastModified, links[], location, referer, title, URL
Form	Action
document.forms []elements[]	checked, defaultChecked, defaultValue, name, selectedIndex, toString, value
History	current, next, previous, toString(), all array elements
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.

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 текущей страницы и 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.

18.2. Контрольные вопросы

1. Что называется литералом?
2. Какой оператор отвечает за определение переменных в JavaScript?
3. На какие виды подразделяются массивы в JavaScript?
4. Назовите ключевое слово, при помощи которого определяют функцию.
5. Что такое объект?
6. Какие есть строковые литералы, зачем они и в чем разница между ними?
7. Какие методы массива вы знаете?
8. За что отвечает метод join()?
9. За что отвечает метод reverse()?
10. За что отвечает метод sort()?
11. Какие есть виды функций?
12. Для чего используют конструктор функций?
13. Что такое логическое выражение?
14. Каковы функции операторов break и continue?
15. За что отвечает оператор return?
16. Что такое фокус?

18.3. Задания

1. Продемонстрируйте использование методов join(), reverse(), sort(). (R1901)
2. Продемонстрируйте возможности функции eval(). (R1902)
3. Напишите примеры использования четырёх операторов языка (операторы выберите самостоятельно). (R1903)
4. Реализуйте скрытую передачу данных из формы. (R1904)
5. Создайте функцию calcD, которая будет возвращать дискриминант квадратного уравнения по формуле b2 – 4ac. (R1905)
6. Напишите функцию приветствия(функция обращается ко внешней переменной). И выведете приветствие на экран (R1906)