Учреждение образования
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Н. А. Жиляк

Введение в JavaScript

*Рекомендовано
учебно-методическим объединением
по образованию в области
информатики и радиоэлектроники
в качестве учебно-методического пособия
для студентов учреждений высшего образования
по направлению специальности 1-40 05 01-03
«Информационные системы и технологии
(издательско-полиграфический комплекс)»*

Минск 2014

УДК 004.434(075.8)
ББК 32.97я73
Ж72

Р е ц е н з е н т ы :
кафедра программного обеспечения информационных технологий Белорусского государственного университета информатики и радиоэлектроники
(доцент, кандидат технических наук, доцент *А. Т. Пешков*);
заведующий кафедрой прикладной математики и информатики ГУО «Институт непрерывного образования» Белорусского государственного университета *Б. В. Лесун*

Жиляк, Н. А.

Ж72 Введение в JavaScript : учеб.-метод. пособие для студентов по направлению специальности 1-40 01 02-03 «Информационные системы и технологии (издательско-полиграфический комплекс)» / Н. А. Жиляк. – Минск : БГТУ, 2014. – 241 с.

ISBN 978-985-530-368-9.

Учебно-методическое пособие содержит основные понятия, принципы организации, описание возможностей языка JavaScript. По всем основным темам предлагаются задания для самостоятельного выполнения. Издание предназначено для студентов, изучающих дисциплину «Основы информационных технологий».

УДК 004.434(075.8)
ББК 32.97я73

ISBN 978-985-530-368-9

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Современный JavaScript – это «безопасный» язык программирования общего назначения. Он не предоставляет низкоуровневых средств работы с памятью, процессором, так как изначально был ориентирован на браузеры, к которым такие требования не предъявляются.

Язык JavaScript позволяет писать различные сценарии (скрипты) на клиентской стороне.

Этот язык является очень простым для изучения, его освоение обязательно для любого web-мастера, причем желательно сразу после CSS и перед PHP, так как настоящие возможности по созданию дизайна сайта раскрываются при объединении HTML, CSS и JavaScript. Это объединение называется DHTML (Dynamic HTML), позволяет создать абсолютно любой внешний вид страниц, с его помощью можно настроить интерактивность страниц.

Применив DHTML при создании сайта, можно поставить определенную точку в его создании, и эта точка является готовой структурой и внешним видом сайта.

Язык JavaScript дает массу возможностей для общения с пользователем без перезагрузки страницы (отсюда и название DHTML). Например, с помощью JavaScript можно делать проверку корректности введенных данных в форму, создавать анимацию на страницах, делая их внешний вид более необычным и захватывающим внимание посетителя.

JavaScript позволяет «на лету» обрабатывать данные из форм, обрабатывать различные действия пользователей (например, нажатие клавиш на клавиатуре, изменения размера окон, движение мыши и так далее), быстро менять стили тегов HTML. Это открывает безграничные просторы в web-дизайне.

В браузере JavaScript есть возможности, которые позволяют выполнять манипуляции со страницей, взаимодействовать с посетителем и, в какой-то мере, с сервером:

- создавать новые HTML-теги, удалять существующие, менять стили элементов, прятать, показывать элементы и т. п.;

- реагировать на действия посетителя, обрабатывать клики мыши, перемещение курсора, нажатие на клавиатуру и т. п.;
- посылать запросы на сервер и загружать данные без перезагрузки страницы (эта технология называется AJAX);
- получать и устанавливать cookie, запрашивать данные, выводить сообщения.

Все эти возможности делают язык JavaScript незаменимым при создании своего web-сайта.

Чтобы почувствовать всю мощь и научиться ее использовать у себя на сайте, необходимо познакомиться с этим языком программирования. Изучив основы JavaScript, Вы сможете писать различные скрипты, позволяющие сделать Вашу страницу максимально удобной и красивой.

Глава 1
ВВОД И ВЫВОД ДАННЫХ. ТИПЫ ДАННЫХ. ОПЕРАТОРЫ

JavaScript (JScript) «родился» в компании Microsoft и предназначается в первую очередь для написания сценариев в HTML страницах. Он очень похож на такие объектно-ориентированные языки, как C++ и Java, однако с помощью JScript вы не сможете создавать самостоятельные приложения, также он ограничен в «общениях» с файлами, зато выигрывает по функциональности и легкости написания web-сценариев (скриптов).

Как и любой другой Script-язык (например, VBScript), JScript помещается в web-страницу по следующим правилам:

- следует корректно размещать сценарии в блоке <*HEAD*>, но IE 4.0 может и «закрыть глаза», если вы так не сделаете;
- сам код размещается между ограничителями <*SCRIPT language="JScript"*> *и* <*/SCRIPT*>. Также, для большей совместимости, можно заключить его в еще одни рамки: <*!-- и //-->*;
- все операторы разделяются точкой с запятой (;);
- комментарии отделяются от программы двумя наклонными чертами (//).

В основном при работе с данным языком создаются функции, отвечающие за работу свойств HTML-объектов, доступ к ним через язык HTML и JScript имеет различный синтаксис.

Действительно, если, например, вы подступаетесь к цвету фона некоторого объекта в HTML-варианте через свойство background-color, то в JScript вам придется писать backgroundColor. Это происходит потому, что символ дефиса в JScript распознается, как оператор вычитания, то есть минус, поэтому его указание в имени свойства приведет к непониманию IE, который этот сценарий будет читать. То же самое происходит и с другими свойствами, имеющими в названии дефис, необходимо убирать его, и делать первую букву второго составляющего слова заглавной.

JScript не имеет огромного разнообразия типов данных. Все данные у него либо строчные, либо целые, либо числа с плавающей

точкой. Причем никакие типы указывать не надо, все переменные определяются одинаково. В JScript переменные определяются несколькими способами:

- предварительно указав ее в начале программы: *var i, J, nameOfTheVariable* и т. д.;
- предварительно указав ее в начале программы и сразу присвоив значение: *var index=10*.

Все разнообразие применения JScript сводится к написанию нескольких функций, поэтому необходимо уметь их определять: пишется function, затем ее имя fnName, потом в скобках указываются доступные аргументы *(arg1, arg2)*, и заключаются все производимые функцией действия в фигурные скобки *{...}*. Для полноты можно указать через закрывающиеся скобки возвращаемое функцией значение в следующей форме: return(значение).

В результате получится:

```
function doSomething (myArg) {
    //... Здесь находится ваш код
    return(myValue);
}
```

Для осуществления доступа к свойствам объектов необходимо знать иерархию их расположения.

Обычно для этого достаточно предварительно указать идентификатор – ID у объекта, с которым намечено производить действие, а затем по нему найти в скрипте объект из множества других.

Все объекты и их свойства находятся в строгом порядке. Сначала идет самый главный объект: *document* (еще можно использовать window, но для других целей), у него есть множество *all*; это множество содержит все объекты, которые существуют в данном документе, т. е. через него возможен доступ к нашему предварительно «меченому» объекту. Когда объект указан, происходит доступ к его свойствам и множествам, например, чтобы изменить цвет объекта, через его множество *style* обращаемся к свойству *color*. Итак, чтобы добраться до свойства, которое надо изменить, надо пройти сквозь всю иерархию объектов. На практике это будет выглядеть следующим образом: *document.all.myObject.style.color='red';*.

Здесь *myObject* – это свойство ID некоторого объекта. В этой связи полезно использовать современный редактор, поддерживающий вывод гиперподсказки с перечислением всех нижеследующих ступеней.

§ 1.1. Правила построения выражений

Важнейшим понятием при изучении основ языка JavaScript является синтаксис (правила построения выражений). Приводимые примеры желательно самостоятельно выполнить на компьютере. Это можно сделать по-разному, на первом этапе рекомендуется воспользоваться самым простым и доступным способом: в качестве интерпретатора (исполнительной системы) программ на JavaScript возьмите веббраузер. В качестве редактора ваших программ выберите какой-нибудь простой текстовый редактор, например, Блокнот Windows. Откройте текстовый редактор и создайте в нем заготовку файла, который вы будете потом редактировать, вводя экспериментальные выражения или даже целые программы. А именно, введите в рабочее поле редактора следующий текст:

```
<HTML>
    <HEAD>
        <TITLE>
            Пример
        </TITLE>
    </HEAD>
    <SCRIPT>
    </SCRIPT>
</HTML>
```

§ 1.2. Ввод и вывод данных

В JavaScript предусмотрены довольно скудные средства для ввода и вывода данных, поскольку он создавался в первую очередь как язык сценариев для вебстраниц. Основой вебстраниц является код, написанный на языке HTML, который специально рассчитан на форматирование информации и создание пользовательского интерфейса. Поскольку сценарии на JavaScript хорошо интегрируются с HTML-кодом, постольку для ввода и вывода данных вполне подойдут средства HTML. Если вы пишете программу на JavaScript, которая будет выполняться веббраузером Internet Explorer, то можете воспользоваться тремя стандартными методами для ввода и вывода данных: *alert()*, *prompt() и confirm()*. Рассмотрим эти методы браузера подробнее.

1.2.1. Метод alert

Данный метод позволяет выводить диалоговое окно с заданным сообщением и кнопкой ОК. Синтаксис соответствующего выражения имеет следующий вид: *alert ("сообщение")*. Если ваше сообщение конкретно, то есть представляет собой вполне определенный набор символов, то его необходимо заключить в двойные или одинарные кавычки. Сообщение представляет собой данные любого типа: последовательность символов, заключенную в кавычки, число (в кавычках или без них), переменную или выражение. Диалоговое окно, выведенное на экран методом *alert()*, можно убрать, щелкнув на кнопке ОК. До тех пор, пока вы не сделаете этого, переход к ранее открытым окнам невозможен. Окна, обладающие свойством останавливать все последующие действия пользователя и программ, называются модальными. Таким образом, окно, создаваемое посредством *alert()*, является модальным (рис. 1.1).

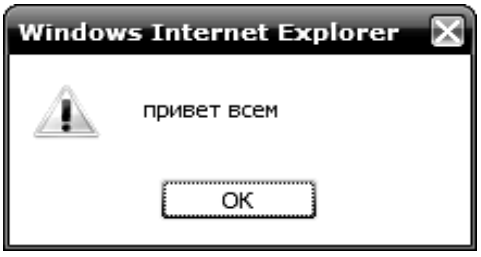

Рис. 1.1. Диалоговое окно,
созданное методом alert ("Привет всем!")

Ранее упоминалось, что метод *a1ert()* можно использовать для вывода промежуточных и окончательных результатов программ при их отладке. При этом можно вывести результат вычисления какого-либо выражения и приостановить дальнейшее выполнение работы программы до тех пор, пока не щелкнете на кнопке ОК.

1.2.2. Метод confirm

Метод confirm позволяет вывести диалоговое окно с сообщением и двумя кнопками – ОК и Отмена (Cancel). В отличие от метода alert этот метод возвращает логическую величину, значение которой зависит от того, на какой из двух кнопок щелкнул пользователь. Если он щелкнул на кнопке ОК, то возвращается значение true (истина, да); если же он щелкнул на кнопке Отмена,

то возвращается значение false (ложь, нет). Возвращаемое значение можно обработать в программе и, следовательно, создать эффект интерактивности, то есть диалогового взаимодействия программы с пользователем. Синтаксис применения метода confirm имеет следующий вид: *confirm("сообщение")*. Если сообщение конкретно, то есть представляет собой вполне определенный набор символов, то его необходимо заключить в кавычки, двойные или одинарные. Например, *confirm("Вы действительно хотите завершить работу?")* (рис. 1.2). Сообщение представляет собой данные любого типа: последовательность символов, заключенную в кавычки, число (в кавычках или без них), переменную или выражение.

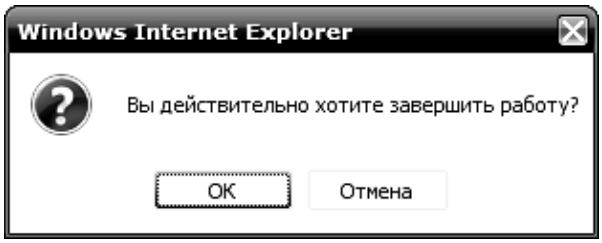

Рис. 1.2. Диалоговое окно,
созданное методом confirm

Диалоговое окно, выведенное на экран методом *confirm()*, можно убрать щелчком на любой из двух кнопок – ОК или Отмена.

1.2.3. Метод prompt

Метод prompt позволяет вывести на экран диалоговое окно с сообщением, а также с текстовым полем, в которое пользователь может ввести данные (рис. 1.3). Кроме того, в этом окне предусмотрены две кнопки: ОК и Отмена (Cancel). В отличие от методов *alert()* и *confirm()* данный метод принимает два параметра: сообщение и значение, которое должно появиться в текстовом поле ввода данных по умолчанию. Если пользователь щелкнет на кнопке ОК, то метод вернет содержимое поля ввода данных, а если он щелкнет на кнопке Отмена, то возвращается логическое значение false (ложь, нет), которое можно затем обработать в программе. Синтаксис применения метода prompt имеет следующий вид: *prompt("сообщение", "значение поля ввода данных")* (рис. 1.3).

Рис. 1.3. Диалоговое окно, созданное методом prompt

Параметры метода *prompt()* не являются обязательными. Если вы их не укажете, то будет выведено окно без сообщения, а в поле ввода данных подставлено значение по умолчанию – undefined (не определено). Если вы не хотите, чтобы в поле ввода данных появлялось значение по умолчанию, то подставьте в качестве значения второго параметра пустую строку *""*. Например, *prompt("Введите Ваше имя, пожалуйста", "")* (рис. 1.4).

Рис. 1.4. Диалоговое окно, созданное методом prompt() без параметра

Диалоговое окно, выведенное на экран методом *prompt()*, можно убрать щелчком на любой из двух кнопок – ОК или Отмена. Как и в случае использования метода *confirm()*, переход к ранее открытым окнам невозможен.

§ 1.3. Типы данных

В любом языке программирования очень важно понятие типа данных (табл. 1.1). Данные, которые хранятся в памяти компьютера и подвергаются обработке, можно отнести к различным типам. Понятие типа данных возникает естественным образом, когда необходимо применить к ним операции обработки. Например, операция умножения применяется к числам, то есть к данным числового типа.

А что получится, если умножить слово «Вася» на число 5? Поскольку трудно дать ответ на этот вопрос, напрашивается вывод: некоторые операции не следует применять к разнотипным данным. Также неизвестно, что должно получиться в результате умножения слов «Вася» и «Маня», поэтому напрашивается вывод о том, что определенные операции вообще не применимы к данным некоторых типов. С другой стороны, существуют операции, результат которых зависит от типа данных.

Таблица 1.1

Типы данных в JavaScript

Тип данных	Примеры	Описание значений
Строковый или символьный (string)	"Привет" "д.т. 123-4567"	Последовательность символов, заключенная в кавычки, двойные или одинарные
Числовой (number)	3.14-567+2.5	Число, последовательность цифр, перед которой может быть указан знак числа (+ или –); перед положительными числами не обязательно ставить знак «+»; целая и дробная части чисел разделяются точкой. Число записывается без кавычек
Логический (булевский, boolean)	true false	true (истина, да) или false (ложь, нет); возможны только два значения
Null		Отсутствие какого бы то ни было значения
Объект (object)		Программный объект, определяемый своими свойствами. В частности, массив также является объектом
Функция (function)		Определение функции – программного кода, выполнение которого может возвращать некоторое значение

Например, операция сложения, обозначаемая символом «+», может применяться и к двум числам, и к двум строкам, состоящим из произвольных слов. В первом случае результатом применения этой операции будет некоторое число, а во втором – строка, получающаяся путем приписывания второй строки к концу первой. В случае строк операцию сложения еще называют склейкой или конкатенацией. Операции, применимые к различным типам данных, но обозначаемые одним и тем же символом, называют также

перегруженными. Так, операция, обозначаемая символом «+», является перегруженной: применительно к числам она выполняет арифметическое сложение этих чисел, а применительно к строкам символов – склейку (приписывание, конкатенацию).

На данном этапе следует обратить особое внимание на различия в представлении числовых и строковых (символьных) данных. Числа как данные числового типа всегда представляются без кавычек. Для их написания мы используем цифры и, при необходимости, знак числа и разделительную точку. Строковые данные заключаются в кавычки, двойные или одинарные. Например, запись *-345.12* представляет число, а запись *"-345.12"* – строку символов. Это данные различных типов, хотя мы и можем сказать, что их содержанием является одно и то же число. Но это не что иное, как обыденная интерпретация данных или, как еще говорят, семантика (смысл) данных. С точки зрения языка программирования число (точнее, данные числового типа) можно корректно использовать в арифметических операциях, а строки – в строковых операциях. Что означает «корректно использовать»? То, что использование данных в языке программирования должно соответствовать нашим традициям, не связанным с программированием. Например, обращение с числами корректно, если оно подчиняется правилам математики; обращение со строками корректно, если не противоречит правилам редакторской правки текста (вставка, удаление, склейка фрагментов и т. п.). Язык программирования призван обеспечить в той или иной мере выполнение операций, имеющих аналоги в обычной человеческой деятельности. Этой цели служит, в частности, и понятие типов данных.

Заметим, что строка, не содержащая ни одного символа (даже пробела), называется пустой. При этом строка, содержащая хотя бы один пробел (например, *" "*), не пуста.

Данные логического типа могут иметь одно из двух значений: true или false. Эти значения записываются без кавычек. Значение true означает истину (да), а false – ложь (нет). Обычно эти значения получаются в результате вычисления выражений с использованием операций сравнения двух данных, а также логических операций (И, ИЛИ, НЕ). Например, результатом вычисления выражения $2 < 3$ является, очевидно, значение true (действительно, число 2 меньше числа 3). Логический тип данных называют еще булевым (boolean)

в честь английского математика Джона Буля, придумавшего алгебру для логических величин.

Другие типы данных (Null, Object и Function) будут рассмотрены в следующих главах, после изучения основ языка.

При создании программ на JavaScript за типами данных следит сам программист. Если он перепутает типы, то интерпретатор не зафиксирует ошибки, а попытается привести данные к некоторому типу, чтобы выполнить указанную операцию. Многие языки программирования, в том числе C и Pascal, не обладают этой особенностью, они требуют явного указания типа данных.

Например, если вы напишете выражение *5+"Вася"*, то результатом его вычисления будет строка символов *"5Вася"*. Таким образом, интерпретатор, столкнувшись с выражением сложения числа и строки символов, переводит число в строку, содержащую это число, а затем выполняет операцию сложения двух строк. Сложение двух строк в JavaScript дает в результате строку, получающуюся путем приписывания второй строки к концу первой.

Результатом вычисления выражения *2+3* будет число *5*, а выражения *2+"3*" – строка *"23"*, состоящая из двух цифровых символов. Как нетрудно заметить, в случае применения операции сложения к числу и строке символов интерпретатор преобразует число в строку символов и возвращает результат вычисления выражения тоже в виде строки символов. Иначе говоря, в случае смысловой несогласованности типов данных интерпретатор использует некоторые правила их согласования, принятые по умолчанию. В результате могут появиться трудно выявляемые ошибки. С другой стороны, эту особенность языка можно использовать для написания изящных и компактных кодов, соблюдая известную осторожность. Для преобразования строк в числа в JavaScript предусмотрены встроенные функции *parseInt()* и *parseFloat()*. Что такое функция, мы подробно расскажем ниже в одном из разделов. А сейчас считайте, что это выражение с круглыми скобками, в которых можно указывать параметры. В результате вычисления функции получается некоторое значение. Функция parseInt(строка, основание) преобразует указанную в параметре строку в целое число в системе счисления по указанному основанию (8, 10 или 16). Если основание не указано, то предполагается 10, то есть десятеричная система счисления.

Примеры

parseInt ("3.14") // результат = 3
parseInt ("-7.875") // результат = -7
parseInt "435") // результат = 435
parseInt ("Вася") / результат = NaN, то есть не является числом*/*
parseInt ("15" ,8) // результат = 13
parseInt ("0xFF",16) // результат =255

Обратите внимание, что при преобразовании в целое число округления не происходит, дробная часть просто отбрасывается.

Функция parseFloat(строка) преобразует указанную строку в число с плавающей разделительной (десятичной, основание) точкой.

Примеры

parseFloat ("3.14") // результат= 3.14
parseFloat("-7.875") // результат = –7.875
parseFloat ("435") // результат = 435
parseFloat ("Вася") / результат = NaN, то есть не является числом*/*
parseFloat ("17.5") // результат = 17.5

Задача преобразования чисел в строки возникает реже, чем обратное преобразование. Чтобы преобразовать число в строку, достаточно к пустой строке прибавить это число, то есть воспользоваться оператором сложения «+». Например, вычисление выражения *""+3.14* даст в результате строку *"3.14"*. Об операторах будет подробно рассказано ниже. Для определения того, является ли значение выражения числом, служит встроенная функция isNaN(значение). Вычисление этой функции дает результат логического типа. Если указанное значение не является числом, функция возвращает true, иначе – false. Однако здесь понятие «число» не совпадает с понятием «значение числового типа». Функция isNaN() считает числом и данные числового типа, и строку, содержащую только число. Логические значения также идентифицируются как числа. При этом значению true соответствует 1, а значению false – 0. Таким образом, если isNaN возвращает false, то это означает, что значение параметра имеет числовой тип, либо является числом, преобразованным в строковый тип, либо является логическим (true или false).

Примеры

isNaN(123) // результат false (то есть это – число)
isNaN ("50 рублей") / результат true (то есть это – не число)*/*

Таблица 1.2

Символы для формирования строковых данных

Символ	Описание
\n	Новая строка
\t	Табуляция
\f	Новая страница
\b	Забой
/r	Возврат каретки

Символы, представленные в табл. 1.2, используются при формировании строковых данных для их последующего отображения. Например, если мы хотим, чтобы сообщение, выводимое на экран в браузере Internet Explorer с помощью функции alert(), отображалось в виде нескольких строк, то следует использовать служебный символ n\ (рис. 1.5):

alert("Фамилия – Иванов \n Имя – Иван \n Отчество – Иванович")

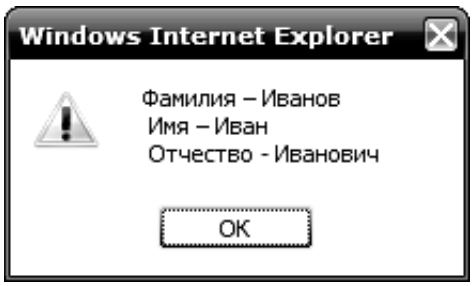

Рис. 1.5. Окно, создаваемое функцией alert(),
в случае, когда строка сообщения
содержит два символа \n

Иногда требуется отобразить символы, имеющие служебное назначение. Как, например, отобразить кавычки, если они используются для задания строки символов? Для этой цели используется «\» (обратная косая черта). Например, чтобы отобразить строку *Акционерное общество "Рога и копыта"* вместе с кавычками,

следует написать такую строку: *"Акционерное общество \"Рога и копыта\""*. Обратная косая черта указывает, что следующий непосредственно за ней символ не нужно интерпретировать как символ синтаксиса языка. В нашем примере она показывает, что кавычки не являются признаком начала или окончания строковых данных, а являются просто элементом этих данных. В Internet Explorer выполнение выражения *alert("Акционерное общество \"Рога и копыта\"")* даст результат, показанный на рис. 1.6.

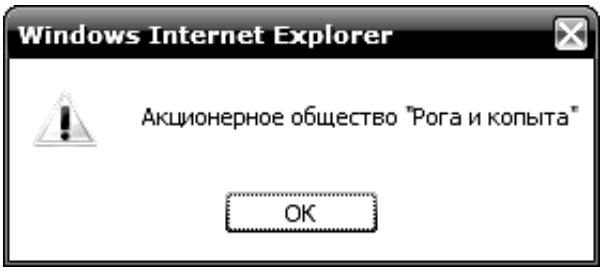

Рис. 1.6. Выполнение выражения alert

Заметим, что эту же задачу можно решить и несколько иначе, используя кавычки различных видов (двойные и одинарные). Во-первых, можно написать так: *'Акционерное общество "Рога и копыта"'*. В этом случае мы использовали одинарные кавычки в качестве признаков начала и конца всей строки. Во-вторых, можно поменять местами кавычки различных видов: *"Акционерное общество 'Рога и копыта'"*. Однако в этом случае название акционерного общества будет отображаться в одинарных кавычках. Наконец, можно внешние кавычки оставить двойными, а внутренние одинарные кавычки продублировать: *"Акционерное общество "Рога и копыта""*. Тогда при отображении строки внутренние кавычки будут заменены на двойные.

Неправильное использование кавычек довольно часто вызывает проблемы у новичков.

ВНИМАНИЕ! Кавычки, обрамляющие строковые данные, должны быть одного вида и использоваться парами.

Интерпретатор, обнаружив в тексте программы кавычки, будет искать еще кавычки такого же вида, считая все, находящееся между ними, строковыми данными.

ВНИМАНИЕ! Внутри строки, заключенной в кавычки одного вида, нужно использовать кавычки другого вида (иначе интерпретатор либо выдаст сообщение об ошибке, либо неправильно воспримет данные).

§ 1.4. Переменные и оператор присвоения

Если в программе просто написать данные какого-либо типа, например число 314, интерпретатор выполнит эту запись, разместив ее во внутреннем формате где-то в памяти компьютера. Чтобы сохранять данные в памяти и в то же время оставлять их доступными для дальнейшего использования, в программах используются переменные.

1.4.1. Имена переменных

Переменную можно считать контейнером для хранения данных. Данные, сохраняемые в переменной, называют значениями этой переменной. Переменная имеет имя – последовательность букв, цифр и символа подчеркивания без пробелов и знаков препинания, начинающуюся обязательно с буквы или символа подчеркивания. Примерами правильных имен переменных являются следующие: *myFamily*, *my_adress*, *_x*, *tel 412_3456*. Примерами неправильных имен переменных – *512group*, *my adress*, *tel:412 3456*. При выборе имен переменных нельзя использовать ключевые слова, то есть слова, используемые в определениях конструкций языка. Например, нельзя выбирать слова var, if, else, const, true, false, function, super, switch и ряд других. Список ключевых слов приведен в параграфе 1.11. Имя должно отражать содержание переменной. Если имя состоит из нескольких слов, то между ними можно вводить символ подчеркивания или писать их слитно, начиная каждое слово с прописной буквы. Вот примеры: *my_first_adress*, *myFirstAdress*. Иногда в качестве первого символа имени используют букву, указывающую на тип данных (значений) этой переменной: c – строковый (character), n – числовой (number), b – логический (boolean), o – объект (object), a – массив (array). Например cAdress, nCena, aMonth. JavaScript является регистрозависимым языком.

Это означает, что изменение регистра символов (с прописных на строчные и наоборот) в имени переменной приводит к другой переменной. Например variable, Variable и vaRiabLe – различные переменные.

СОВЕТ! Выработайте для себя правила образования имен, которые не должны противоречить указанным выше требованиям. Если им следовать, то уменьшится вероятность ошибок в ваших программах.

1.4.2. Создание переменных

Создать переменную в программе можно несколькими способами:

- с помощью оператора присвоения значений в формате: имя_переменной = значение. Оператор присвоения обозначается символом равенства «=»;
- с помощью ключевого слова var (от variable – переменная) в формате: var имя_переменной. В этом случае созданная переменная не имеет никакого значения, но может его получить в дальнейшем с помощью оператора присвоения. Пример: *var myName myName = "Иван"*;
- с помощью ключевого слова var и оператора присвоения в формате: var имя_переменной = значение.

Тип переменной определяется типом значения, которое имеет строка. В отличие от многих других языков программирования, при инициализации переменной не нужно описывать ее тип. Переменная может иметь значения различных типов и неоднократно их изменять. Например, можно написать следующий код программы

```
var x = "Иван"
// некоторый код
x = "Анна"
// некоторый код
x = 2.5
```

Одно ключевое слово var можно использовать для инициализации сразу нескольких переменных, как с оператором присвое-

ния, так и без него. При этом переменные и выражения с операторами присвоения разделяются запятыми, например

var name = "Вася", address, x = 3.14

Если переменная в данный момент имеет значение числового типа, то имеется в виду, что это числовая переменная. Аналогично можно говорить о строковых, логических (булевых), неопределенных (в случае типа null) переменных. Выше использовался оператор присвоения значения переменной, обозначаемый символом равенства «=». Не следует путать этот оператор с отношением равенства и соответствующей операцией сравнения. Выражение с оператором «=» интерпретатор вычисляет следующим образом: переменной слева от него присваивается значение, расположенное справа от него. Если x и y – две переменные, то выражение $x = y$ интерпретируется так: переменной x присваивается значение переменной y. Иначе говоря, переменной можно присвоить значение другой переменной, указав справа от оператора «=» ее имя. Таким образом, к значениям можно получать доступ опосредованно – через имена переменных.

1.4.3. Область действия переменных

Переменные, которые созданы в программе с помощью оператора присвоения с использованием ключевого слова var или без него, являются глобальными. Это означает, что они доступны всюду в этой же программе, а также в вызываемых программах из других файлов. Эти переменные также доступны внутри кода функций. Кроме глобальных существуют и локальные переменные. Они могут быть созданы внутри кода функций. Можно определить переменные с одинаковыми названиями и во внешней программе, и в коде функции. В этом случае переменные, определенные в коде функции с помощью ключевого слова var, будут локальными, т. е. изменения их значений внутри функции никак не отразятся на переменных с такими же именами, определенных во внешней программе. При этом значения локальных переменных не доступны из внешней программы. Необходимо отметить, что в программировании это понятие играет очень важную роль. Нередко область действия называют областью видимости. Переменная может быть видна или не видна внутри программной единицы (функции, подпрограммы). Область видимости, доступности или

действия переменной – эквивалентные термины. Кроме них еще используют понятие времени жизни переменной. Время жизни переменных в JavaScript определяется интервалом времени от загрузки до выгрузки программы из памяти компьютера. Так, если программа (сценарий) записаны в HTML-коде веб-страницы, то после его выгрузки весь сценарий вместе с определенными в нем переменными прекращает активное существование.

§ 1.5. Операторы

Операторы предназначены для составления выражений. Операнд в языках программирования – аргумент операции; данные, которые обрабатываются командой; грамматическая конструкция, обозначающая выражение, задающее значение аргумента операции. Например, оператор сложения применяется к двум операндам, а оператор логического отрицания – к одному операнду. Элементарное выражение, состоящее из операндов и оператора, вычисляется интерпретатором и, следовательно, имеет некоторое значение. В этом случае говорят, что оператор возвращает значение. Например, оператор сложения, примененный к числам 2 и 3, возвращает значение 5. Оператор имеет тип, совпадающий с типом возвращаемого им значения. Поскольку элементарное выражение с оператором и операндами возвращает значение, это выражение можно присвоить переменной. Один оператор мы уже рассмотрели в предыдущем параграфе. Это оператор присвоения «=». Мы часто будем его использовать. Однако следует заметить, что существует еще пять разновидностей оператора присвоения, сочетающих в себе действия обычного оператора «=» и операторов сложения, вычитания, умножения, деления и деления по модулю. Эти дополнительные операторы присвоения мы рассмотрим позже в параграфе 1.5.3.

1.5.1. Комментарии

Операторы комментариев позволяют выделить фрагмент программы, который не выполняется интерпретатором, а служит лишь для пояснений содержания программы. В JavaScript допустимы два вида операторов комментария:

// – одна строка символов, расположенная справа от этого оператора, считается комментарием;

/*...*/ – все, что заключено между /* и */, считается комментарием; с помощью этого оператора можно выделить несколько строк в качестве комментария.

СОВЕТ! Не пренебрегайте комментариями в тексте программ. Это поможет при их отладке и сопровождении. На этапе разработки лучше сначала превратить ненужный фрагмент программы в комментарий, чем просто удалить (а вдруг его придется восстанавливать?).

Даже опытные программисты, возвращаясь к работе над своей программой через месяц, с большим трудом вспоминают, что к чему. В примерах программ мы часто будем использовать комментарии.

1.5.2. Арифметические операторы

Арифметические операторы, такие как сложение, умножение и т. д., в JavaScript могут применяться к данным любых типов (табл. 1.3).

Таблица 1.3

Арифметические операторы

Оператор	Название	Пример
+	Сложение	*X* + *Y*
-	Вычитание	*X* - *Y*
*	Умножение	*X* * *Y*
/	Деление	*X* / *Y*
%	Деление по модулю	*X* % *Y*
++	Увеличение на 1	*X*++
--	Уменьшение на 1	*Y*--

Арифметические операторы лучше всего рассматривать на примере чисел. Первые четыре оператора все проходили в начальных классах. Оператор деления по модулю возвращает остаток отделения первого числа на второе. Например, 8%3 возвращает 2. Операторы «+» и «-» сочетают в себе действия операторов соответственно сложения и вычитания, а также присвоения.

Выражение *X++* эквивалентно выражению *X+1*, а выражение *X--* – выражению *X-1*. Операторы «++» и «--» называют соответственно инкрементом и декрементом. Символ «-» используется не только как бинарный (для двух операндов) оператор сложения, но и как унарный (для одного операнда) оператор отрицания для указания того, что число является отрицательным.

Однако формально ничто не мешает нам применить эти операторы к данным других типов. В случае строковых данных оператор сложения дает в результате строку, полученную путем приписывания справа второй строки к первой. Например, выражение *"Вася"+"Маша"* возвращает в результате *"ВасяМаша"*. Поэтому для строк оператор сложения называется оператором склейки или конкатенации. Применение остальных арифметических операторов к строкам дает в результате NaN – значение, не являющееся числом. В случае, когда оператор сложения применяется к строке и числу, интерпретатор переводит число в соответствующую строку и далее действует как оператор склейки двух строк.

В случае логических данных интерпретатор переводит логические значения операндов в числовые (true в 1, false в 0), выполняет вычисление и возвращает числовой результат. То же самое происходит в том случае, когда один оператор логический, а другой – числовой.

Если один операнд строкового типа, а другой – логического, то в случае сложения интерпретатор переводит оба операнда в строковый тип и возвращает строку символов, а в случае других арифметических операторов он переводит оба операнда в числовой тип.

На первый взгляд, применение арифметических операторов к разнотипным данным может показаться довольно запутанным. Необходимо учитывать следующие правила:

- применение арифметических операторов возможно к данным одного и того же типа;
- оператор сложения применительно к строкам действует как оператор их склейки (приписывания, конкатенации);
- в случае применения арифметических операторов к логическим данным интерпретатор рассматривает значения true и false как числа соответственно 1 и 0, и возвращает результат в числовом виде.

Кроме этих правил нужно помнить, что в JavaScript имеются средства для преобразования типов данных, то есть для преобразования данных одного типа в данные другого типа.

1.5.3. Дополнительные операторы присвоения

В начале этой главы уже упоминалось, что кроме обычного оператора присвоения «=» имеются еще пять дополнительных операторов, сочетающих в себе действия обычного оператора присвоения и арифметических операторов.

Таблица 1.4

Операторы присвоения

Оператор	Пример	Эквивалентное выражение
+=	*x+=y*	*x=x+y*
-=	*x-=y*	*x=x-y*
*=	*x*=y*	*x=x*y*
/=	*x/=y*	*x=x/y*
%=	*x%=y*	*x=x%y*

Поскольку действие арифметических операторов и обычного оператора присвоения уже известны, дополнительные операторы присвоения здесь подробно не рассмотрены. Обращается лишь внимание на экономный способ записи выражений, предоставляемый этими операторами.

1.5.4. Операторы сравнения

В программах часто приходится проверять, выполняются ли какие-либо условия. Например, на веб-страницах иногда проверяется, является ли браузер пользователя Microsoft Internet Explorer или Netscape Navigator. В зависимости от результата процесс дальнейших вычислений может пойти по тому или другому пути. Проверяемые условия формируются на основе операторов сравнения, таких как «равно», «меньше», «больше» и т. п. Результатом вычисления элементарного выражения, содержащего оператор сравнения и операнды (сравниваемые данные), является логическое значение, то есть true или false. Так, если условие выполняется (верно, справедливо), то возвращается true. В противном случае возвращается false.

ВНИМАНИЕ! Оператор «равно» записывается с помощью двух символов без пробелов между ними.

Сравнивать можно числа, строки и логические значения. Сравнение чисел происходит по правилам арифметики, а строк – путем сравнения ASCII-кодов символов, начиная с левого конца строк. Логические значения сравниваются так же, как и числа 1 и 0 (true соответствует 1, а false – 0). Как видим, с числами и логическими данными все довольно просто. А вот результат сравнения строк не всегда очевиден (табл. 1.5).

Таблица 1.5

Операторы сравнения

Оператор	Название	Пример
= =	Равно	*X*= =*Y*
!=	Не равно	*X*!=*Y*
>	Больше, чем	*X*>*Y*
>=	Больше или равно (не меньше)	*X*>=*Y*
<	Меньше, чем	*X*<*Y*
<=	Меньше или равно (не больше)	*X* <= *Y*

Здесь не приводится таблица кодов ASCII для всех символов. Отмечается лишь, что некоторые из них упорядочены по возрастанию ASCII-кодов следующим образом: сначала идет пробел, затем в порядке возрастания цифры от 0 до 9, символ подчеркивания, латинские и кириллические буквы по алфавиту (сначала прописные, а затем строчные). Заметим, что операторы сравнения могут быть применены и к разнотипным данным. Если сравниваются строка и число, то интерпретатор приводит операнды к числовому типу. То же самое происходит при сравнении логических данных и числа. Если сравниваются логические данные и строка, то дело обстоит несколько сложнее. В этом случае результат не зависит от содержимого строки. Если она содержит число (но не цифры с буквами), только пробелы или является пустой, то операнды приводятся к числовому типу. При этом пустая строка ( ) или содержащая только пробелы преобразуется в число 0. В остальных случаях все операторы сравнения, кроме !=, будут возвращать false (а оператор «!=» – противоположный результат, то есть true).

1.5.5. Логические операторы

Логические данные, обычно получаемые с помощью элементарных выражений, содержащих операторы сравнения, можно

объединять в более сложные выражения. Для этого используются логические (булевские) операторы – логические союзы И и ИЛИ, а также оператор отрицания НЕ. Например, нам может потребоваться сформировать сложное логическое условие следующего вида: «возраст не более 30 и опыт работы больше 10, или юрист». В этом примере есть и операторы сравнения, и логические операторы. Выражения с логическими операторами возвращают значение true или false (табл. 1.6).

Таблица 1.6

Логические операторы

Оператор	Название	Пример
!	Отрицание (НЕ)	!*X*
&&	И	*X*&&*Y*
||	ИЛИ	*X*||*Y*

Оператор отрицания «!» применяется к одному операнду, изменяя его значение на противоположное: если *X* имеет значение *true*, то !*X* возвращает значение *false*, и наоборот, если *X* имеет значение *false*, то !*X* возвращает значение *true*. Ниже в таблице указано, какие значения возвращают операторы И и ИЛИ при различных логических значениях двух операндов (табл. 1.7).

Таблица 1.7

Оператор отрицания

X	*Y*	*X*&&*Y*	*X*||*Y*
true	true	true	false
true	false	false	true
false	true	false	true
false	false	false	true

Операторы «&&» и «||» еще называют логическим умножением и логическим сложением соответственно. Если вспомнить, что значению true можно сопоставить 1, а значению false – 0, то нетрудно понять, как вычисляются значения элементарных выражений с логическими операторами. Нужно только учесть, что в алгебре логики 1 + 1 = 1 (а не 2). Аналогично оператору отрицания соответствует вычитание из единицы числового эквивалента логического

значения операнда. В математике логические операции И и ИЛИ называют соответственно конъюнкцией и дизъюнкцией.

СОВЕТ! Чтобы не запутаться, не применяйте логические операторы к нелогическим данным и особенно к разнотипным данным.

Сложные логические выражения, состоящие из нескольких более простых, соединенных операторами И и ИЛИ, выполняются по так называемому принципу короткой обработки. Дело в том, что значение всего выражения иногда можно определить, вычислив лишь одно или несколько более простых выражений, не вычисляя остальные. Например, выражение *x&&y* вычисляется слева направо; если значение x оказалось равным false, то значение y не вычисляется, поскольку и так известно, что значение всего выражения равно false. Аналогично, если в выражении *x* || *y* значение *x* равно true, то значение y не вычисляется, поскольку уже ясно, что все выражение равно true. Данное обстоятельство требует особого внимания при тестировании сложных логических выражений. Так, если какая-нибудь составная часть выражения содержит ошибку, то она может остаться невыявленной, поскольку эта часть выражения просто не выполнялась при тестировании.

1.5.6. Операторы условного перехода

Вычислительный процесс можно направить по тому или другому пути в зависимости от того, выполняется ли некоторое условие или нет. Этой цели служат операторы условного перехода if и switch.

Оператор if. Оператор условного перехода if позволяет реализовать структуру условного выражения если ..., то ..., иначе ...

Синтаксис оператора if перехода следующий:

```
if (условие)
{ код, который выполняется, если условие выполнено}
else
{ код, который выполняется, если условие не выполнено}
```

В фигурных скобках располагается блок кода – несколько выражений. Если в блоке используется не более одного выражения, то фигурные скобки можно не писать. Часть этой конструкции, определяемая ключевым словом else (иначе), необязательна.

В этом случае остается только часть, определенная ключевым словом if (если).

if (условие)
{ код, который работает, если условие выполнено}

Конструкция оператора условного перехода допускает вложение других операторов условного перехода. Условие обычно представляет собой выражение логического типа, то есть выражение, значение которого есть true или false. Обычно это элементарные выражения с операторами сравнения.

Оператор switch. Для организации проверки большого количества условий вполне достаточно использовать рассмотренный выше оператор if. Однако в случае нескольких условий более удобным и наглядным оказывается оператор switch (переключатель). Он особенно удобен, если требуется проверить несколько условий, которые не являются взаимоисключающими. Синтаксис оператора switch выглядит следующим образом:

switch (выражение)
{
case вариант:
код
[break]
case вариант2:
код
[break]
[default:
код]
}

Параметр «выражение» оператора *switch* может принимать строковые, числовые и логические значения. Разумеется, в случае логического выражения возможны только два варианта. Ключевые слова (операторы) *break* и *default* не являются обязательными, что отражено с помощью прямоугольных скобок. Здесь они являются элементами описания синтаксиса, и при написании операторов их указывать не нужно. Если оператор *break* указан, то проверка остальных условий не производится. Если указан оператор *default*, то следующий за ним код выполняется, если значение выражения

не соответствует ни одному из вариантов. Если все варианты возможных значений предусмотрены в операторе switch, то оператор *default* можно не использовать.

Оператор *switch* работает следующим образом. Сначала вычисляется выражение, указанное в круглых скобках сразу за ключевым словом *switch*. Полученное значение сравнивается с тем, которое указано в первом варианте. Если они не совпадают, то код этого варианта не выполняется и происходит переход к следующему варианту. Если же значения совпали, то выполняется код, соответствующий этому варианту. При этом, если не указан оператор *break*, то выполняются коды и остальных вариантов, пока не встретится оператор *break*. Это же правило действует и для остальных вариантов.

1.5.7. Операторы цикла

Оператор цикла обеспечивает многократное выполнение блока программного кода до тех пор, пока не выполнится некоторое условие. В JavaScript предусмотрены три оператора цикла: *for*, *while* и *do-while*. Вообще говоря, при создании программ вполне можно обойтись одним из них, *fo*r или *while*. Однако возникают ситуации, в которых один из операторов более удобен или естествен, чем другой.

Оператор for. Оператор *for* (для) также называют оператором со счетчиком циклов, хотя в нем совсем не обязательно использовать счетчик. Синтаксис этого оператора следующий:

```
for ( [начальное выражение] ; [условие] ; [выражение обновления] )
{
код
}
```

Здесь квадратные скобки лишь указывают на то, что заключенные в них параметры не являются обязательными. Как и в случае оператора условного перехода, возможна и такая запись:

```
for ( [начальное выражение] ; [условие] ; [выражение обновления] ) {
   код
}
```

Все, что находится в круглых скобках справа от ключевого слова for, называется заголовком оператора цикла, а содержимое фигурных скобок – его телом. В заголовке оператора цикла начальное выражение выполняется только один раз в начале выполнения оператора. Второй параметр представляет собой условие продолжения работы оператора цикла. Он аналогичен условию в операторе условного перехода *if*. Третий параметр содержит выражение, которое выполняется после выполнения всех выражений кода, заключенного в фигурные скобки. Оператор цикла работает следующим образом. Сначала выполняется начальное выражение. Затем проверяется условие. Если оно выполнено, то оператор цикла прекращает работу (при этом код не выполняется). В противном случае выполняется код, расположенный в теле оператора *for*, то есть между фигурными скобками. После этого выполняется выражение обновления (третий параметр оператора for). Таким образом, заканчивается первый цикл или, как еще говорят, первая итерация цикла. Далее снова проверяется условие, и все повторяется описанным выше способом. Обычно в качестве начального выражения используют оператор присваивания значения переменной. Например, *i = 0* или var *i = 0*. Имя переменной и присваиваемое значение могут быть любыми. Эту переменную называют счетчиком циклов. В этом случае условие, как правило, представляет собой элементарное выражение сравнения переменной счетчика циклов с некоторым числом, например, *i <= nMax*. Выражение обновления в таком случае просто изменяет значение счетчика циклов, например *i = i + 1* или, короче, *i++*. В следующем примере оператор цикла просто изменяет значение своего счетчика, выполняя 15 итераций

```
for (i = 1; i <= 15; i++) {}
```

Немного модифицируем этот код, чтобы вычислить сумму всех целых положительных чисел от 1 до 15:

```
var s = 1
for (i = 1; i <= 15; i++)
{
s = s + i
}
```

Заметим, что счетчик циклов может быть не только возрастающим, но и убывающим.

Типовая структура оператора цикла с использованием *break* имеет следующий вид

```
for ( [начальное выражение] ; [условие!] ; [выражение обновления] )
{
Код
 if (условие2){
 Код
 break }
 код
}
```

Для управления вычислениями в операторе цикла можно также использовать оператор *continue* (продолжение). Также, как и *break*, этот оператор применяется в теле оператора цикла вместе с оператором условного перехода. Однако, в отличие от break, оператор continue прекращает выполнение последующего кода, выполняет выражение обновления и возвращает вычислительный процесс в начало оператора цикла, где производится проверка условия, указанного в заголовке. Типовая структура оператора цикла с использованием *break* имеет следующий вид

```
for ( [начальное выражение] ; [условие!] ; [выражение обновления] )
{
код
if (условие2){
код
continue
}
 код
}
```

Оператор while. Оператор цикла *while* (до тех пор, пока) имеет структуру более простую, чем оператор *for*, и работает несколько иначе. Синтаксис этого оператора следующий

```
while ( условие )
{
код
}
```

При выполнении этого оператора сначала производится проверка условия, указанного в заголовке, то есть в круглых скобках справа от ключевого слова *while*. Если оно выполняется, то выполняется код в теле оператора цикла, заключенного в фигурные скобки. В противном случае код не выполняется. При выполнении кода (завершении первой итерации) вычислительный процесс возвращается к заголовку, где снова проверяется условие, и т. д. Если сравнивать оператор *while* с оператором *for*, то особенность первого заключается в том, что выражение обновления записывается в теле оператора, а не в заголовке. Часто забывают указать это выражение, и в результате цикл не завершается (программа «зависает»). Рассмотрим несколько примеров решения задач, которые уже решали ранее с использованием оператора цикла for.

Пример. Возведение *x* в степень *y*.

```
/* Вычисляем x в степени y */
var z = x
i = 2
while (i = 2) {
z =z*x
i++
}
```

Оператор do-while. Оператор *do-while* (делай до тех пор, пока) представляет собой конструкцию из двух операторов, используемых совместно. Синтаксис этой конструкции следующий

```
do {
код
}
while (условие)
```

В отличие от оператора *while* в операторе *do-while* код выполняется хотя бы один раз, независимо от условия. Условие проверяется после выполнения кода. Если оно истинно, то снова выполняется код в теле оператора *do*. В противном случае работа оператора *do-while* завершается. В операторе *while* условие проверяется в первую очередь, до выполнения кода в теле. Если при первом обращении к оператору *while* условие не выполняется, то код не будет выполнен никогда. Рассмотрим несколько примеров решения задач, которые мы уже решали ранее с использованием операторов цикла *for* и *while*.

Пример. Возведение *x* в степень *y*.

```
/* Вычисляем x в степени y */
var z = x
i = 2
do{
z=z*x
i++
} while (i <= y)
```

1.5.8. Выражения с операторами

Выше неоднократно встречались термины «выражение» и «элементарное выражение». Вы, должно быть, уже составили себе некоторое представление о том, что такое выражение. Тем не менее, уточним его. Начнем с того, что просто данные (конечные значения) являются выражениями языка. Например, число 5.2, одиноко стоящее в строке текста программного кода, является выражением. Последовательность символов, заключенная в кавычки (например, *"Вася"*), – тоже выражение. Имя переменной – еще один вариант выражения. Запись, содержащая имя переменной, за которой следуют символы оператора присвоения и некоторое значение (например, *x = "Привет всем!"*), является выражением JavaScript. Запись, состоящая из операндов и оператора (например, *x+5*), также является выражением. Все перечисленные выше варианты выражения назовем элементарными выражениями. Тогда записи, содержащие операторы и операнды в виде элементарных выражений, являются выражениями. Например, пусть имеются два элементарных выражения: *x+5* и *y-3*. Тогда запись *x+5 + y-3*, объединяющая их оператором сложения, является выражением. Это касается не только арифметических операторов, но и всех других. Итак, можно писать сложные выражения, в которых операторы различных типов встречаются несколько раз. Выражение с последовательностью из нескольких операторов вычисляется слева направо, но с учетом так называемого приоритета операторов.

Процессом вычисления выражений можно управлять с помощью круглых скобок. Каждой открывающейся скобке соответствует своя закрывающаяся скобка, так что выражения, заключенные в круглые скобки, могут входить в состав других выражений, которые, в свою очередь, также могут быть заключены в скобки. Таким образом, с помощью круглых скобок можно создать иерархическую

структуру выражения, состоящего из других выражений. Первыми выполняются выражения, имеющие наибольшую глубину вложенности. Далее выполняются выражения, находящиеся на меньшей глубине в иерархии, и т. д. Выражения с одинаковой глубиной вложенности выполняются в порядке, принятом по умолчанию, то есть слева направо.

Операторы условного перехода и цикла также представляют собой выражения языка JavaScript. Кроме обычных способов их записи, рассмотренных выше, принципиально возможна запись в одну строку. При этом необходимо, чтобы выражения в блоках кода разделялись точкой с запятой.

Пример

if (!x){x = "Вы ничего не ввели"; alert (x) }else alert("Все в порядке")

Операторы условного перехода и цикла возвращают значения подобно другим операторам, а именно возвращают значение последнего выполненного выражения. Сводные сведения о приоритетах всех операторов приведены в параграфе 1.10 в конце данной главы.

§ 1.6. Функции

Функция представляет собой подпрограмму, которую можно вызвать для выполнения, обратившись к ней по имени. Взаимодействие функции с внешней программой, из которой она была вызвана, происходит путем передачи функции параметров и приема от нее результата вычислений. Впрочем, функция в JavaScript может и не требовать параметров, а также ничего не возвращать.

В JavaScript есть встроенные функции, которые можно использовать в программах, но код которых нельзя редактировать или посмотреть. Все, что мы можем узнать о них, – это описание их действия, параметров и возвращаемого значения. Кроме использования встроенных функций можно создать свои собственные, так называемые пользовательские функции. Часто используемые фрагменты программного кода целесообразно оформлять в виде функций. Такой фрагмент кода заключается в фигурные

скобки, а перед ним пишется ключевое слово function, за которым следуют круглые скобки, обрамляющие список параметров. Чтобы вызвать функцию в программе, следует написать выражение в следующем формате

имя_функции (параметры)

Если требуются параметры, то они указываются в круглых скобках через запятую. Функция может и не иметь параметров. В этом случае в круглых скобках ничего не указывается. Подробности использования функций изложены далее в этом параграфе.

1.6.1. Встроенные функции

В JavaScript имеются нижеследующие встроенные функции (некоторые из них мы уже рассматривали ранее). Хотя для иллюстрации работы этих функций приводится множество примеров, желательно выполнить их самим, а также придумать свои примеры, обращая внимание на крайние (даже абсурдные) случаи.

parseInt(строка, основание) – преобразует указанную строку в целое число в системе счисления по указанному основанию (8, 10 или 16); если основание не указано, то предполагается 10, то есть десятеричная система счисления.

parseFloat(строка, основание) – преобразует указанную строку в число с плавающей разделительной (десятичной) точкой в системе счисления по указанному основанию (8, 10 или 16); если основание не указано, то предполагается 10, то есть десятеричная система счисления.

isNaN(значение) – возвращает true, если указанное в параметре значение не является числом, иначе – false.

Перечисленные выше функции мы рассматривали в параграфе 1.3, посвященном типам данных.

eval(строка) – вычисляет выражение в указанной строке; выражение должно быть написано на языке JavaScript (не содержит тегов HTML).

1.6.2. Пользовательские функции

Пользовательские функции – это функции, которые можно создать самим, по своему усмотрению, для решения своих задач. Функция задается своим определением (описанием), которое

начинается ключевым словом function. Точнее, описание функции имеет следующий синтаксис:

function имя_функции(параметры){
код
}

Часто определение функции записывают и в таких формах:

function имя_функции(параметры) { код }

Имя функции выбирается также, как и имя переменной. Недопустимо использовать в качестве имени ключевые слова языка JavaScript. За именем функции обязательно стоит пара круглых скобок. Программный код (тело) функции заключается в фигурные скобки. Они определяют группу выражений, которые относятся к коду именно этой функции. Если функция принимает параметры, то список их имен (идентификаторов) указывается в круглых скобках около имени функции. Имена параметров выбираются согласно тем же требованиям, что и имена обычных переменных. Если параметров несколько, то в списке они разделяются запятыми. Если параметры для данной функции не предусмотрены, то в круглых скобках около имени функции ничего не пишут. Когда создается определение функции, список ее параметров (если он необходим) содержит просто формальные идентификаторы (имена) этих параметров, понимаемые как переменные. В определении функции в списке параметров, заключенном в круглые скобки сразу же за именем функции после ключевого слова function, нельзя использовать конкретные значения и выражения. В этом смысле определение функции задает код, оперирующий формальными параметрами, которые конкретизируются лишь при вызове функции из внешней программы. Если требуется, чтобы функция возвращала некоторое значение, то в ее теле используется оператор возврата return с указанием справа от него того, что следует возвратить. В качестве возвращаемой величины может выступать любое выражение: простое значение, имя переменной или вычисляемое выражение. Оператор return может встречаться в коде функции несколько раз. Впрочем, возвращаемую величину, а также сам оператор return можно и не указывать. В этом случае функция ничего не будет возвращать.

Пример

Допустим, требуется определить функцию для вычисления площади прямоугольника (для этого необходимо умножить ширину на высоту). Назовем эту функцию Srectangle. Тогда ее определение будет выглядеть следующим образом

```
function Srectangle(width, height)
{
 S = width * height
return S
}
```

Здесь сначала вычисляется площадь путем умножения значений параметров width и height, полученное значение присваивается переменной *S*, а затем оператор return возвращает значение этой переменной.

Определение функции Srectangle можно сделать более экономным, минуя присвоение значения переменной *S*

```
function Srectangle(width, height){
return width * height
}
```

Определение функции, описанное выше, будучи помещенным в программу, усваивается интерпретатором, но сама функция (ее код или тело) не выполняется. Чтобы выполнить функцию, определение которой задано, необходимо написать в программе выражение вызова этой функции. Оно имеет следующий синтаксис:

имя_функции (параметры)

Имя функции должно полностью (вплоть до регистра) совпадать с именем ранее определенной функции. Параметры, если они заданы в определении функции, в вызове функции представляются конкретными значениями, переменными или выражениями.

ВНИМАНИЕ! Не путайте определение функции с ее вызовом, хотя и то и другое может находиться в одной и той же программе.

В JavaScript можно не поддерживать равенство между количествами параметров в определении функции и в ее вызове. Если в функции определены, например, три параметра, а в вызове указаны только

два, то последнему параметру будет автоматически присвоено значение null. Наоборот, лишние параметры в вызове функции будут просто проигнорированы. Выше мы рассмотрели пример определения функции Srectangle(), вычисляющей площадь прямоугольника, если заданы его ширина и высота. Чтобы вычислить значение площади прямоугольника при конкретных значениях параметров, в программе необходимо написать выражение вызова функции Srectangle(), указав в качестве ее параметров конкретные значения, либо переменные с конкретными значениями, либо выражения, вычисляющие значения.

Определение функции может содержать в себе определения других функций, однако такие вложенные функции доступны только из кода функции, содержащей их определения. Определение функции создает так называемый экземпляр объекта Function, обладающий полезными свойствами и методами. В частности, он имеет свойство arguments, содержащее информацию о параметрах, которые действительно были переданы функции при ее вызове. В некоторых случаях, прежде чем выполнить выражения тела функции, требуется проанализировать значения параметров. И тогда понадобится использовать свойство arguments. Подробности описаны ниже, в параграфе 1.7.7.

1.6.3. Выражения с функциями

Выше были рассмотрены выражения с операторами. К множеству элементарных выражений JavaScript можно добавить еще и вызовы функций. Тогда вызов функции может быть правой частью оператора присвоения, а также составной частью любого выражения с операторами.

Примеры

1. Пусть *Srectangle(width, height)* – функция, возвращающая значение площади прямоугольника со сторонами width и height. Тогда для вычисления площади прямоугольного треугольника с катетами *a* и *b* можно использовать следующее выражение:

S3 = 0.5 Srectangle(a, b)*

В этом примере вызов функции является операндом выражения с арифметическим оператором умножения. При этом значение выражения присваивается переменной S3.

2. Вызов функции может использоваться в логических выражениях:

```
if (Srectangle(a, b) > Srectangle(c, d))
alert("Первый прямоугольник больше второго")
```

3. В качестве параметра функции может указываться вызов другой функции:

```
var x = "25px"
var y = 12
var S = Srectangle(parseInt(x) , y)
```

В выражениях с вызовами функций последние имеют наивысший приоритет.

Глава 2
ВСТРОЕННЫЕ ОБЪЕКТЫ

§ 2.1. Встроенные объекты

Объекты представляют собой программные единицы, обладающие некоторыми свойствами. Об объекте можно судить по значениям его свойств и описанию того, как он функционирует. Программный код встроенных в JavaScript объектов нам недоступен. Главное для нас сейчас – усвоить, как пользоваться объектами. Это совсем просто, нужно только соблюдать нехитрый синтаксис. Использовать объекты не труднее, чем функции. Управление веб-страницами с помощью сценариев, написанных на JavaScript, заключается в использовании и изменении свойств объектов HTML-документа и самого браузера. Эти вопросы мы отложим до следующих глав.

Встроенные объекты имеют фиксированные названия и свойства. Все свойства этих объектов разделяют на два вида: просто свойства и методы. Свойства аналогичны обычным переменным. Они имеют имена и значения. Некоторые свойства объектов доступны только для чтения. Это означает, что их значения нельзя изменять. Другие свойства доступны и для записи – их значения можно изменять с помощью оператора присвоения. Методы аналогичны функциям, они могут иметь параметры или не иметь их. Чтобы узнать значение свойства объекта, необходимо указать имя этого объекта и имя свойства, отделив их друг от друга точкой: имя_объекта.свойство. Заметим, что объект может и не иметь свойств. Мы можем заставить объект выполнить тот или иной присущий ему метод. В этом случае также говорят о применении метода к объекту. Синтаксис соответствующего выражения такой: имя_объекта.метод(параметры). Заметим, что объект может не иметь методов. Итак, по синтаксису свойства отличаются от обычных переменных тем, что имеют составные имена, а также тем, что значения некоторых свойств нельзя измененить. Методы отличаются с точки зрения синтаксиса от обычных функций только

тем, что имеют составные имена. В свете изложенного выше объект можно понимать как некоторый контейнер, содержащий переменные-свойства и функции-методы. Разумеется, в этом контейнере есть еще что-то, но оно скрыто от нас. Есть возможность воздействовать на объект только с помощью свойств и методов. В некоторых источниках часто сравнивают объект с черным ящиком, у которого есть входы и выходы, доступные для наблюдения и, возможно, для управления. В JavaScript математические вычисления, сложная обработка строк и дат, а также создание массивов производятся с помощью соответствующих встроенных объектов. Для разработчиков веб-сайтов особенно важны объекты String (обработка строк), Array (массивы), Math (математические формулы и константы) и Date (работа с датами). Обратите на них особое внимание. Встроенные объекты, как уже отмечалось, имеют фиксированные названия. Объекты с именами, совпадающими с их фиксированными названиями, называются статическими. Однако можно создать экземпляры (копии) статических объектов, присвоив им свои собственные имена. Экземпляры статических объектов являются объектами в вашей программе, которые наследуют от первых все их свойства и методы. Экземпляры объектов – это некоторые частные воплощения в программе соответствующих статических объектов. Вместе с тем можно использовать и статические объекты в чистом виде, не создавая никаких их копий. Например, для формульных вычислений используется статический объект Math, а в случае массивов создаются экземпляры статического объекта Array, содержащие конкретные данные, к которым применимы все общие методы и свойства статического объекта Array.

Встроенные объекты имеют, среди прочих, свойство prototype (прототип), с помощью которого можно добавлять новые свойства и методы к уже существующим экземплярам объектов. Эти новые свойства и методы, разумеется, необходимо предварительно самим продумать и воплотить в виде программных кодов. Ниже рассмотрим, как это делается. Например, при желании можно создать свой собственный метод обработки строк или массивов и присоединить их к конкретным объектам, чтобы затем использовать так же, как и встроенные свойства и методы. При разработке сценариев для веб-страниц такая задача редко возникает, но JavaScript предназначен не только для создания сценариев.

2.1.1. Объект String (Строка)

Объект String представляет интерес главным образом благодаря методам обработки строк. Он незаменим, когда требуется, например, найти позицию вхождения одной строки в другую, вырезать из строки некоторую ее часть, разбить строку на отдельные элементы и создать из них массив и т. д.

С помощью объекта String можно создать строку как строковый объект. Однако в подавляющем большинстве случаев для этого достаточно использовать обычную переменную и оператор присвоения строкового значения. В этом случае интерпретатор все равно создает экземпляр (копию) строкового объекта, свойства и методы которого доступны из программного кода.

Создание строкового объекта. Для создания строкового объекта используется выражение следующего вида:

имя_переменной = new String("строковое_значение")

Здесь имя_переменной выполняет роль ссылки на строковый объект. Например, выражение *mystring = new String ("Привет!")* создает строковый объект mystring со значением *"Привет!"*.

Однако можно создать строковый объект и с помощью обычного оператора присвоения:

имя_переменной = "строковое_значение"
или
var имя_переменной = "строковое_значение"

Доступ к свойствам и методам строкового объекта обеспечивается такими выражениями:

строка.свойство
String.свойство
строка.метод([параметры])
String.метод([параметры])

Некоторые методы могут и не иметь параметров, что указано с помощью квадратных скобок. Здесь строка может быть ссылкой на строковый объект, строковой переменной, выражением, возвращающим строку, а также просто строковым значением.

Когда используется ключевое слово String в качестве имени объекта, это означает, что нас интересуют свойства и методы статического строкового объекта, то есть общие свойства и методы, не связанные, вообще говоря, с конкретными свойствами и методами конкретного строкового объекта (экземпляра объекта String). Ниже приведены три различных способа использования свойства length строкового объекта, значением которого является длина строки (количество символов в строке).

```
mystring = "Однажды в студеную зимнюю пору"
mystring.length // значение равно 30
"Однажды в студеную зимнюю пору".length // значение равно 30
function fstring() {return "abcde"} /* функция, возвращающая строку "abcde" */
fstring().length //значение равно 5
```

Методы строкового объекта используются для синтаксической обработки и форматирования строк. Эти две группы методов мы рассмотрим отдельно.

Свойства String. length – длина или, иными словами, количество символов (включая пробелы) в строке; целое число.

Методы String обработки строк. Метод charAt(индекс) – возвращает символ, занимающий в строке указанную позицию.

Синтаксис: *строка. charAt (индекс)*

Возвращает односимвольную или пустую строку.

Параметр (индекс) является числом, индекс первого символа равен 0.

Примеры

```
"Привет".charAt(2) // значение равно "и"
"Привет".charAt(15) // значение равно ""
mystring = "Привет" mystring.charAt(mystring.length-1) /* значение последнего символа равно "т" */
```

Метод *charCodeAt([индекс])* – преобразует символ в указанной позиции строки в его числовой эквивалент (код).

Синтаксис: *строка. charCodeAt([индекс])*

Возвращает число. IE4+ и NN6 поддерживают систему кодов Unicode, NN4 – ISO-Latin 1.

Примеры

"abc" . charCodeAt() // значение равно 97
"abc" . charCodeAt(1) // значение равно 98
"abc" . charCodeAt(25) // значение равно NaN
"" . charCodeAt(25) // значение равно NaN
"я". charCodeAt(0) // значение равно 1103

Метод fromCharCode(номерl [, номер2 [, ... номерN]]) – возвращает строку символов, числовые коды которой указаны в качестве параметров.

Синтаксис: *String.fromCharCode(номерl [, номер2 [,... номерN]])*

Возвращает строку. IE4+ и NN6 поддерживают систему кодов Unicode, NN4 – ISO-Latin 1.

Пример

String.fromCharCode(97,98,1102) // значение равно "abю"

Метод concat(строка) – конкатенация (склейка) строк.

Синтаксис: строка1.concat(строка2)

Возвращает строку.

Этот метод действует так же, как и оператор «+» сложения для строк: к строке *строка1* приписывается справа *строка2*.

Метод indexOf(строка_поиска [, индекс]) – производит поиск строки, указанной параметром, и возвращает индекс ее первого вхождения.

Синтаксис: *строка.indexOf(строка_поиска [, индекс])*

Возвращает число.

Метод производит поиск позиции первого вхождения *строка_поиска* в строку *строка*. Возвращаемое число (индекс вхождения) отсчитывается от 0. Если поиск неудачен, то возвращается -1. Поиск в пустой строке всегда возвращает -1. Поиск пустой строки всегда возвращает 0.

Второй параметр, не являющийся обязательным, указывает индекс, с которого следует начать поиск.

Этот метод хорошо использовать вместе с методом выделения подстроки substr() (см. ниже), когда требуется сначала определить позиции начала и конца выделяемой подстроки. Рассматриваемый здесь метод подходит для определения начальной позиции.

Примеры

```
x = "Во первых строках своего письма"
x.indexOf("первых") // значение равно 3
x.indexOf("первых строках") // значение равно 3
x.indexOf("вторых строках") // значение равно -1
x.indexOf("В") // значение равно 0
x.indexOf ("в") // значение равно 6
x.indexOf("в" , 7) // значение равно 19
x.indexOf(" ") // значение равно 2
x.indexOf(" ", 5) // значение равно 9
x.indexOf("") // значение равно 0
```

Метод LastIndexOf(строка_поиска [, индекс]) – производит поиск строки, указанной параметром, и возвращает индекс ее первого вхождения; при этом поиск начинается с конца исходной строки, но возвращаемый индекс отсчитывается от ее начала, то есть от 0.

Синтаксис: *строка.LastIndexOf(строка_поиска [, индекс])*

Возвращает число. Метод аналогичен рассмотренному выше indexOf() и отличается лишь направлением поиска.

Примеры

```
x ="Во первых строках своего письма"
x.lastIndexOf("первых") // значение равно 3
x.lastIndexOf("а") // значение равно 30
x.indexOf("а") // значение равно 15
```

Метод 1oca1eCompare(строка) – позволяет сравнивать строки в кодировке Unicode, то есть с учетом используемого браузером языка общения с пользователем.

Синтаксис: *строка1. 1oca1eCompare (строка2)*

Возвращает число. Совместимость: IE5.5+, NN6+.

Если сравниваемые строки одинаковы, метод возвращает 0. Если строка1 меньше, чем строка2, то возвращается отрицательное число, в противном случае – положительное. Сравнение строк происходит путем сравнения сумм кодов их символов. Абсолютное значение возвращаемого числа зависит от браузера. Так, IE5.5 и IE6.0 возвращают 1 или -1, а NN6 – разность сумм кодов в кодировке Unicode.

Метод slice(индекс1 [, индекс2]) – возвращает подстроку исходной строки, начальный и конечный индексы которой указываются параметрами, за исключением последнего символа.

Синтаксис: *строка.slice(индекс1 [, индекс2])*

Возвращает строку. Данный метод не изменяет исходную строку.

Если второй параметр не указан, то возвращается подстрока с начальной позицией и до конца строки. Отсчет позиций начинается с начала строки. Первый символ строки имеет индекс0. Если второй параметр указан, то возвращается подстрока исходной строки, начиная с позиции индекс1 и до позиции индекс2, исключая последний символ. Если второй параметр отрицателен, то отсчет конечного индекса производится от конца строки. В этом заключается основное отличие метода slice() от substr(). Сравните также этот метод с методом substring().

Метод split(разделитель [, ограничитель]) – возвращает массив элементов, полученных из исходной строки.

Синтаксис: *строка.split(разделитель [, ограничитель])*

Возвращает массив.

Первый параметр является строкой символов, используемой в качестве разделителя строки на элементы. Второй необязательный параметр – число, указывающее, сколько элементов строки, полученной при разделелении, следует включить в возвращаемый массив.

Если разделитель – пустая строка, то возвращается массив символов строки.

Примеры

```
x = "Привет всем"
x.split(" ")    /* значение – массив из элементов: "Привет", "всем" */
x.split("е")    /* значение – массив из элементов: "Прив", "твс", "м" */
x.split("е",2)  /* значение – массивиз элементов: "Прив", "т вс" */
```

Метод substr(индекс [, длина]) – возвращает подстроку исходной строки, начальный индекс и длина которой указываются параметрами.

Синтаксис: *substr(индекс [, длина])*

Возвращает строку. Данный метод не изменяет исходную строку.

Если второй параметр не указан, то возвращается подстрока позицией и до конца строки. Отсчет позиций начинается с начала строки.

Первый символ строки имеет индекс 0. Если второй параметр указан, то возвращается подстрока исходной строки, начиная с позиции индекс1 и с общим количеством символов, равным длина. Сравните этот метод с методами и substring().

Метод substring(индекс1 индекс2) – возвращает подстроку исходной строки, начальный и конечный индексы которой указываются параметрами.

Синтаксис: *substring (индекс1 индекс2)*

Возвращает строку. Данный метод не изменяет исходную строку.

Порядок индексов не важен: наименьший из них считается начальным. Отсчет позиций начинается с начала строки. Первый символ строки имеет индекс 0. Символ, соответствующий конечному индексу, не включается в возвращаемую строку. Сравните этот метод с методами *substring()* и *slice()*.

Примеры

```
x = "Привет всем"
x.substring(0 , 6) // значение равно "Привет"
x. substring(7 , x. length) // значение равно "всем"'
x. substring(7, 250) // значение равно "всем"
x. substring(250, 7) // значение равно "всем"
```

Методы tolocaleLowerCase(), toLowerCase() – переводят строку в нижний регистр.

Синтаксис: *tolocaleLowerCase(), toLowerCase()*

Возвращают строку. Первый метод работает в IE5.5+, NN6, учитывает определенные языковые системы.

Приведение строк к одному и тому же регистру требуется, например, при сравнении с одержимого строк без учета регистра. Кроме того, многие серверы чувствительны к регистру, в котором определены имена файлов и папки (обычно требуется, чтобы они были определены в нижнем регистре).

Методы toLocaleUpperCase(), toUpperCase() – переводят строку в верхний регистр.

Синтаксис: *строка.toLocaleUpperCase(), строка.tolUpperCase()*

Возвращают строку. Первый метод работает в IE5.5+, NN6, учитывает определенные языковые системы.

Приведение строк к одному и тому же регистру требуется, например, при сравнении содержимого строк без учета регистра. Кроме того, многие серверы чувствительны к регистру, в котором определены имена файлов и папки.

Функции вставки и замены подстрок. При обработке строк часто требуется вставить или заменить подстроки. Удаление подстроки из данной строки можно рассматривать как частный случай замены, а именно замены указанной подстроки пустой строкой. С помощью методов объекта String можно написать программу для решения этой задачи. Поскольку она часто встречается, то целесообразно оформить программу в виде функций.

Рассмотрим сначала функцию вставки строки в исходную строку. Назовем ее, например, insstr. Данная функция должна принимать три параметра: исходную строку s1, вставляемую строку s2 и индекс позиции вставки *n*. Ниже приведены ее определение и примеры вызова:

```
function insstr(s1.s2,n)
{
ireturnsl.slice(0,n) + s2 + s1.slice(n)
}
insstr("Привет, друзья", " мои", 7) // "Привет, мои друзья"
insstr(привет, друзья", " мои",  100)     // "Привет, друзья мои"
```

Функции удаления передних и заключительных пробелов. При обработке строк (например, введенных пользователем в поля формы) нередко возникает задача удалить лишние передние и задние пробелы. Многие языки программирования имеют соответствующие встроенные функции, но в JavaScript их нет. Поэтому не помешает создать их самим с помощью имеющихся средств и поместить в свою библиотеку (сохранить в текстовом файле с расширением .js).

Решить поставленную задачу можно несколькими способами. Рассмотрим следующий алгоритм. Преобразуем исходную строку в массив слов, используя в качестве разделителя один пробел. Затем необходимо проанализировать первые или последние элементы массива в зависимости от того, что нам требуется: удалить передние или задние пробелы в исходной строке. Допустим,

необходимо удалить передние пробелы. Тогда, если исходная строка содержала *N* пробелов, первые *N* элементов массива будут содержать пустые отроки *""*. Мы проверяем в цикле значения первых элементов, пока не найдем непустой элемент или пока не исчерпаем весь массив. Если первый непустой элемент массива имеет индекс, то создадим новый массив, содержащий элементы исходного, начиная с этого индекса. Наконец, склеим элементы этого массива в одну строку, используя в качестве разделителя строку с единственным пробелом *" "*. Все, что понадобится, – это методы объекта String, оператор цикла и оператор условного перехода.

2.1.2. Объект Array (Массив)

Массив представляет собой упорядоченный набор данных. Его удобно представить себе в виде одностолбцовой таблицы, содержащей некоторое количество строк. В ячейках такой таблицы могут находиться данные любого типа, в том числе и массивы. В последнем случае можно говорить о многомерных массивах (то есть о массивах массивов). Количество элементов в массиве (строк в таблице) называется длиной массива. К элементам массива можно обращаться в программе по их порядковому номеру (индексу). Нумерация элементов массива начинается с нуля, так что первый элемент имеет индекс 0, а последний – на единицу меньший, чем длина массива.

Массивы применяются во многих более или менее сложных программах обработки данных, а в некоторых случаях без них просто не обойтись. Если среди используемых данных есть группы таких, которые обрабатываются одинаковым образом, то, возможно, лучше организовать их в виде массива.

Создание массива. Существует несколько способов создания массива. В любом случае прежде всего создается новый объект массива с использованием ключевого слова new:

имя_массива = new Array([длина_массива])

Здесь длина_массива является необязательным числовым параметром, о чем говорят квадратные скобки. Если длина массива не указана, то создается пустой массив, не содержащий ни одного элемента. В противном случае создается массив с указанным количеством элементов, однако все они имеют значение null (то есть не имеют значений).

Можно сначала создать пустой массив, а затем добавить к нему нужное количество элементов с помощью оператора присвоения. Заметим, что выражение с ключевыми словами new Array создает экземпляр (копию) объекта Array.

У объекта Array имеется свойство length, значением которого является длина массива. Чтобы получить значение этого свойства, необходимо использовать выражение

имя_массива.length

Создав массив, можно присвоить значения его элементам, используя для этого оператор присвоения. В левой части оператора присвоения указывается имя массива, а рядом с ним в квадратных скобках индекс элемента. Мы уже говорили, что к элементам массива обращаются по индексу: имя_массива[индекс]. Здесь квадратные скобки обязательны.

Рассмотрим создание массива earth, содержащего в качестве элементов некоторые характеристики нашей планеты. Обратите внимание, что элементы в этом массиве различных типов (строковые и числовые).

```
earth = new Array(4) // массив из 4-х элементов
earth[0] = "Планета" earth[1]   =   "24 часа"
earth[2] = 6378
earth[3] = 365.25
earth.length // значение равно 4
```

Если нам потребуется значение, например, третьего элемента массива, то достаточно использовать выражение earth.

Другой способ создания массива заключается в непосредственном определении элементов в круглых скобках за ключевым словом Array.

Многомерные массивы. Массивы, рассмотренные выше, являются одномерными. Их можно представить себе в виде таблицы из одного столбца. Однако элементы массива могут содержать данные различных типов, в том числе и объекты, а значит, и массивы. Если в качестве элементов некоторого одномерного массива создать массивы, то получится двухмерный массив. Обращение к элементам такого массива происходит в соответствии со следующим синтаксисом:

имя_массива[индекс_уровня1] [индекс_уровня2]

Если массив имеет размерность, большую двух, то синтаксис обращения к массивам аналогичен: следует добавить нужное количество квадратных скобок, заключающих нужные индексы.

Типичным примером двухмерного массива является массив опций меню. У такого меню есть горизонтальная панель с опциями, называемая главным меню. Некоторым опциям главного меню соответствуют раскрывающиеся вертикальные подменю со своими опциями. Мы создаем массив, длина которого равна количеству опций главного меню. Элементы этого массива определяем как массивы названий опций соответствующих подменю. Чтобы была ясна структура нашей конструкции, мы выбрали названия опций надлежащим образом. Например, "Меню 2.1" – название 1-й опции подменю, соответствующего 2-й опции главного меню.

```
menu = new Array()
imenu[0] = new array ("Меню 1.1", "Меню 1.2", "Меню 1.3")
imenu[l] = new Array ("Меню 2.1", "Меню 2.2")
menu[2] = new Array ("Меню 3.1", "Меню 3.2" , "Меню 3.3", "Меню 3.4")
```

Чтобы обратиться ко 2-й опции 3-го подменю, следует написать:

```
menu[2][l] // значение равно "Меню 3.2"
```

Усложним нашу конструкцию, чтобы она содержала не только названия опций подменю, но и названия опций главного меню:

```
menu = new Array() /* Массив опций главного меню: */
menu[0] = new Array ("Меню1", "Меню2", "Меню3")
menu[l] = new Array ()
menu[l][0] = new Array («Меню 1.1". "Меню 1.2", "Меню 1.3")
menu[l] [1] = new Array («Меню 2.1", "Меню 2.2")
menu[l][2] = new Array («'Меню 3.1", "Меню 3.2" , "Меню 3.3", "Меню 3.4")
menu[0][l] // значение равно "Меню 2"
menu[0][2] // значение равно "Меню 3"
menu[l] [1] [0] // значение равно "Меню 2.1"
menu[l] [2] [3] // значение равно "Меню 3.2"
```

Копирование массива. Иногда требуется создать копию массива, чтобы сохранить исходные данные и предохранить их от последующих модификаций. Например, метод сортировки элементов массива, который рассмотрен ниже, изменяет исходный массив. Однако для копирования массива недостаточно присвоить его другой переменной. Используя новую переменную и оператор присвоения, мы создаем лишь новую ссылку на прежний массив, а не новый массив.

Пример

```
a = new Array(5,  2, 4,  3)
x = a // ссылка на массив a
a[2]   = 25 // изменение значения элемента с индексом 2
x[2] // значение равно 25, то есть новому значению
a[2]
```

В этом примере массивы *a* и *x* совпадают.

Чтобы скопировать массив, то есть создать новый массив, элементы которого равны соответствующим элементам исходного, следует воспользоваться оператором цикла, в котором элементам нового массива присваиваются значения элементов исходного, например:

```
a = new Array(5, 2, 4, 3)
x = new Array() // ссылка на массив a
for(1=0;  i<a.length;   i++) /* копирование значений массива a
в элементы массива x */
jx[i] = a [i] }
```

Свойства Array.

1. **length** – длина или, иными словами, количество элементов в массиве; целое число.

Синтаксис: *имя_массива.length*

Поскольку индексация элементов массива начинается с нуля, индекс последнего элемента на единицу меньше длины массива. Это обстоятельство удобно использовать при добавлении к массиву нового элемента: *myarray[mayarray.length] = значение*.

2. **prototype** – свойство (прототип), позволяющее добавить новые свойства и методы ко всем созданным массивам.

```
myarray = new Array() // создание массива myarray
xarray = new Array() // создание массива xarray
```

Array.prototype.author = "Иванов" / добавление прототипа ко всем массивам */*

myarray.author = "Иванов младший" / изменение свойства author для myarray */*

xarray.author = "Сидоров" / изменение свойства author для xarray */*

Методы Array. Методы объекта Array предназначены для управления данными, сохраненными в структуре массива.

1. **concat(Массив)** – конкатенация массивов, объединяет два массива в третий массив.

Синтаксис: *имя_массива1. concat (массив2)*

Возвращает массив. Данный метод не изменяет исходные массивы.

Пример

A1 = new array(1, 2, "Звезда") a2 = new array("а", "б", "в", "г")

a3 = a1.concat(a3) / результат – массив с элементами: 1, 2, "Звезда", "а", "б", "в", "г" */*

2. **join(разделитель)** – создает строку из элементов массива с указанным разделителем между ними; является строкой символов (возможно, пустой).

Синтаксис: *имя_массива.join(строка)*

Возвращает строку символов.

Примеры

a = new array(1, 2, "Звезда")
a.join(",") // значение – строка "1,2,Звезда"
a = new arrayd(1, 2, "Звезда")
a.join(" ") // значение – строка "1 2 Звезда"

3. **pop()** – удаляет последний элемент массива и возвращает его значение. Синтаксис: *имя_массива.pop()*

Возвращает значение удаленного элемента массива. Совместимость: IE5.5+. Данный метод изменяет исходный массив.

4. **push(значение|объект)** – добавляет к массиву указанное значение в качестве последнего элемента и возвращает новую длину массива.

Синтаксис: *имя_массива1. push (значение|объект)*

Возвращает число. Совместимость: IE5.5+. Данный метод изменяет исходный массив.

5. **shift()** – удаляет первый элемент массива и возвращает его значение. Синтаксис: *имя_массива.shift()*

Возвращает значение удаленного элемента массива. Совместимость: IE5.5+, NN4+. Данный метод изменяет исходный массив.

6. **unshift (значение|объект)** – добавляет к массиву указанное значение в качестве первого элемента.

Синтаксис: *имя_массива. unshift (значение|объект)*

Возвращает: ничего. Совместимость: IE5.5+. Данный метод изменяет исходный массив.

7. **reverse()** – изменяет порядок следования элементов массива на противоположный.

Синтаксис: *имя_массива. reverse ()*

Возвращает массив. Данный метод изменяет исходный массив.

Пример

a = new array (1, 2, "Звезда")
a. reverse() / массив с элементами в следующем порядке: "Звезда", 2, 1 */*

8. **sline(индекс1 [, индекс2])** – создает массив из элементов исходного массива с индексами указанного диапазона.

Синтаксис: *имя_массива. sline (индекс1 [, индекс.2])*

Возвращает массив. Данный метод не изменяет исходный массив.

Второй параметр (конечный индекс) не является обязательным, о чем свидетельствуют квадратные скобки в описании синтаксиса. Если он не указан, то создаваемый массив содержит элементы исходного массива, начиная с индекса и до конца. В противном случае создаваемый массив содержит элементы исходного массива, начиная с индекса и до индекса индекс2, за исключением последнего. При этом исходный массив остается без изменений.

Пример

a = new array(1, 2, "Звезда","a" , "b")
a.slise(1,3) // массив с элементами: 2, "Звезда"
a.slise(1) // массив с элементами: "Звезда", "a", "b"

9. **sort([функция_сортировки])** – сортирует (упорядочивает) элементы массива с помощью функции сравнения.

Синтаксис: *имя_массива. sort ([функция_сравнения])*

Возвращает массив. Данный метод изменяет исходный массив. Параметр не обязателен, о чем свидетельствуют квадратные скобки.

Если параметр не указан, то сортировка производится на основе сравнения ASCII-кодов символов значений. Это удобно для сравнения символьных строк, но не совсем подходит для сравнения чисел. Так, число 357 при сортировке считается меньшим, чем 85, поскольку сначала сравниваются первые символы и только в случае их равенства сравниваются следующие, и т. д. Таким образом, метод без параметра подходит для простой сортировки массива со строковыми элементами.

Можно создать свою собственную функцию для сравнения элементов массива, с помощью которой метод sort() отсортирует весь массив. Имя этой функции (без кавычек и круглых скобок) передается методу в качестве параметра. При работе метода функции передаются два элемента массива, а ее код возвращает методу значение, указывающее, какой из элементов должен следовать за другим. Допустим, сравниваются два элемента, *x* и *y*. Тогда в зависимости от числового значения (отрицательного, 0 или положительного), возвращаемого функцией сравнения, методом принимается одно из трех возможных решений (табл. 2.1).

Таблица 2.1

Значения, возвращаемые функцией сравнения

Значения, возвращаемые функцией сравнения	Результат сравнения *x* и *y*
<0	*y* следует за *x*
0	Порядок следования *x* и *y* не изменится
>0	*x* следует за *y*

Итак, по какому критерию сортировать элементы массива, определяется кодом функции сравнения. Если элемент массива имеет значение null, то в Internet Explorer он размещается в начале массива.

Пример

myarray = new Array(4, 2, 15, 3, 30) // числовой массив
function comp(x, y) { // функция сравнения return x-y }
myarray.sort(comp) / массив с элементами в порядке: 2, 3, 4. 15, 30 */*

10. **splite(индекс, количество [элем1, [, элем2 [,)** – удаляет из массива несколько элементов и возвращает массив из удаленных элементов или заменяет значения элементов.

Синтаксис: *имя_массива.splite(индекс, количество [, элем1 [, элем2 [, ...элемN]]])*

Возвращает массив. Совместимость: IE5.5+. Данный метод изменяет исходный массив.

Первые два параметра обязательны, а следующие – нет. Первый параметр является индексом первого удаляемого элемента, а второй – количеством удаляемых элементов.

Пример

a = new Array("Вася", "Иван", "Марья", 12, 5)
x = a.spliced,3) / x – массив элементов: "Иван", "Марья", 12 a – массив элементов: "Вася", 5 */*

Метод позволяет также заменить значения элементов исходного массива, если указаны третий и, возможно, последующие параметры. Эти параметры представляют значения, которыми следует заменить исходные значения элементов массива. При таком использовании метода splice() важен первый параметр (индекс), а второй (количество) может быть равным нулю. В любом случае, если количество элементов замены больше значения второго параметра, то часть элементов исходного массива будет заменена, а часть элементов будет просто вставлена в него. При этом метод возвращает другой массив, состоящий из элементов исходного, индексы которых соответствуют первому и второму параметрам. Но это справедливо, если второй параметр не равен 0.

Пример

a = new Array("Вася", "Иван", "Марья", 12, 5)
x = a.spliced,3,"Петр", "Кузьма", "Анна") // x – массив элементов: "Иван", "Марья", 12 / a – массив элементов: "Вася","Петр", Кузьма", "Анна",5 */*

```
a = new Array("Вася", "Иван", "Марья", 12, 5)
x = a.splice( 1,0, "Петр", "Кузьма", "Анна", "Федор", "Ханс") // x – пустой массив
/* a – массив элементов:
"Вася", "Петр", "Кузьма", "Анна", "Федор", "Ханс" */
```

11. **toLocaleString(), toString()** – преобразуют содержимое массива в символьную строку.

Метод toLocaleString() поддерживается браузерами IE5.5+ и NN3+, а метод toString() – и более ранними версиями. Алгоритм преобразования по методу зависит от версии браузера.

Для преобразования содержимого массива в строку рекомендуется использовать метод join().

2.1.3. Объект Number (Число)

При разработке веб-страниц математические операции используются не столь часто, в отличие от строковых. Обычно они связаны с изменением координат элементов страницы (свойства top, left, width, height таблицы стилей). Однако встречаются и более сложные случаи. Например, может потребоваться вычислить статистические характеристики данных, содержащихся в некоторой таблице. Так или иначе, в этих задачах необходимо иметь дело с числами. О числах мы уже говорили в параграфе, посвященном типам данных. Теперь рассмотрим их более подробно.

Числа в JavaScript. В JavaScript числа могут быть только двух типов: целые и с плавающей точкой. Целые числа не имеют дробной части и не содержат разделительной точки. Числа с плавающей точкой имеют целую и дробную части, разделенные точкой.

Операции с целыми числами процессор компьютера выполняет значительно быстрее, чем операции с числами, имеющими точку. Это обстоятельство имеет смысл учитывать, когда расчетов много. Например, индексы, длины строк являются целочисленными. Число π, многие числа, полученные с помощью оператора деления, денежные суммы и т. п. являются числами с плавающей точкой.

В JavaScript можно производить операции с числами различных типов. Это очень удобно. Однако при этом следует знать, какого числового типа будет результат. Если результат операции является числом с дробной частью, то он представляется как число

с плавающей точкой. Если результат оказался без дробной части, то он приводится к целочисленному типу, а не представляется числом, у которого в дробной части одни нули.

Примеры

2+3 //7 – целое число
2 + 3.6 // 5.6 – число с плавающей точкой
2.4 + 3.6 // 6 – целое число
6.00 // число с плавающей точкой

Числа в JavaScript можно представлять в различных системах счисления, то есть в системах с различными основаниями: 10 (десятичной), 16 (шестнадцатеричной) и 8 (восьмеричной). К десятичной форме представления чисел мы привыкли, однако следует помнить, что числа в этой форме не должны начинаться с нуля, потому что так записываются числа в восьмеричной системе.

Запись числа в шестнадцатеричной форме начинается с префикса 0x (или 0X), где первый символ ноль, затем следуют символы шестнадцатеричных цифр: 0 (ноль), 1, 2, 9, a, b, c, d, e, f (буквы могут быть в любом регистре). Например, шестнадцатеричное число 0x4af в десятеричном представлении есть 1199.

Запись числа в восьмеричной форме начинается с нуля, за которым следуют цифры от 0 до 7. Например, 027 (в десятеричном представлении – 23). В арифметических выражениях числа могут быть представлены в любой из перечисленных выше систем счисления, однако результат всегда приводится к десятеричной форме.

Создание объекта Number. Числа можно создавать обычным образом с помощью переменных и оператора присвоения, не прибегая к объекту Number. Однако этот объект обладает некоторыми полезными свойствами и методами, которые иногда могут пригодиться.

Объект Number создается с помощью выражения вида:

переменная = new Number(число)

Доступ к свойствам и методам строкового объекта обеспечивается такими выражениями:

число.свойство
Number.свойство
число.метод([параметры])
Number.метод([параметры])

Свойства Number

• MAX_VALUE – константа, значение которой равно наибольшему допустимому в JavaScript значению числа (1.7976931348623157e+308).

• MIN_VALUE – константа, значение которой равно наименьшему допустимому в JavaScript значению числа (5e-324).

• NEGATIVE.JNFINITY – число, меньшее, чем Number.MIN VALUE.

• POSITTVEJNFINITY – число, большее, чем Number.MAX_VALUE.

• NaN – константа, имеющая значение NaN, посредством которой JavaScript сообщает, что данные (параметр, возвращаемое значение) не являются числами (Not a Number).

• prototype – свойство (прототип), играющее такую же роль, что и в случае объекта String (см. выше).

Методы Number. Объект Number имеет несколько методов, из которых мы рассмотрим только четыре, предназначенные для представления чисел в виде строки в том или ином формате.

1. toExponential(количество) – представляет число в экспоненциальной форме.

Синтаксис: *число. toExponential (количество)*

Возвращает строку. Совместимость: IE5.5+, NN6+.

Параметр представляет собой целое число, определяющее, сколько цифр после точки следует указывать.

Примеры

```
Number(456)
x.toExponential(3) x.toExponential(2) x.toExponential(1)
x.toExponential(0)
//, 4.560e+2 // 4.56e+2 // 4.6e+2 // 5e+2
```

2. toFixed(количество) – представляет число в форме с фиксированным количеством цифр после точки.

Синтаксис: *число. toFixed (количество)*

Возвращает строку. Совместимость: IE5.5+, NN6+.

Параметр представляет собой целое число, определяющее, сколько цифр после точки следует указывать.

Примеры

x = new Number(25.65)
x.toFixed(3) // 25.658
x.toFixed(2) // 25.65
x.toFixed(l) //25.7
x.toFixed(0) //25.7

3. toPredsion(точность) – представляет число с заданным общим количеством значащих цифр.

Синтаксис: *число. toPredsion (точность)*

Возвращает строку. Совместимость: IE5.5+, NN6+.

Параметр представляет собой целое число, определяющее, сколько всего цифр, до и после точки, следует указывать.

Примеры

X = new Number(135.45)
x.toPrecision(6) // 135.450
x.toPrecision(5) // 135.45
x.toPrecision(4) // 135.5
x.toPrecision(3) // 135
x.toPrecision(2) // 1.4e2
x.toPrecision(l) // 1e2
x.toPrecision(0) // Сообщение об ошибке

4. toString([основание]) – возвращает строковое представление числа в системе счисления с указанным основанием.

Синтаксис: *число. toString ([основание])*

Возвращает строку. Если параметр не указан, имеется в виду десятеричная система счисления. Можно указать 2 для двоичной системы или 16 – для шестнадцатеричной. Заметим, что этот метод имеют все объекты.

Примеры

x = new Number(127.18)
x.toString() // "127.18"
x.toString(10) // "127.18"
x.toString(16) // "7f .2el47ael47b"
x.toString(8) //"177.134121727024366"
x = new Number(5)
x.toString(2)// "101"

2.1.4. Объект Math (Математика)

Объект Math предназначен для хранения некоторых математических констант (например, число π) и выполнения преобразований чисел с помощью типичных математических функций. Доступ к свойствам и методам объекта Math обеспечивается следующими выражениями:

Math.свойство
Math.метод(параметры)

Пример

Для вычисления длины окружности при известном радиусе требуется число, которое можно взять как свойство объекта Math.

```
var R = 10 // радиус окружности
circus = 2*R*Math.PI // длина окружности
```

Методы Math

• abc(число) – возвращает модуль (абсолютное значение) числа;
• acos(число) – возвращает арккосинус числа;
• asin(число) – возвращает арксинус числа;
• atan(число) – возвращает арктангенс числа;
• atan2(*x*, *y*) – возвращает угол в полярных координатах точки;
• ceil(число) – округляет число вверх до ближайшего целого;
• cos(число) – возвращает косинус числа;
• exp(число) – возвращает число *e* в степени *число*;
• floor(число) – округляет число вниз до ближайшего целого;
• log(число) – возвращает натуральный логарифм числа;
• max(число1,число2) – возвращает большее из чисел *число1*, *число2*;
• min(число1,число2) – возвращает меньшее из чисел *число1*, *число2*;
• pow(число1,число2) – возвращает *число1* в степени *число2*;
• random () – возвращает случайное число между 0 и 1;
• round(число) – округляет число до ближайшего целого;
• sin(число) – возвращает синус числа;
• sqrt(число) – возвращает квадратный корень из числа;
• tan(число) – возвращает тангенс числа.

2.1.5. Объект Date (Дата)

Во многих приложениях приходится отображать дату и время, подсчитывать количество дней, оставшихся до заданной даты, и т. п. Некоторые программы даже управляются посредством значений дат и времени. В основе всех операций, связанных с датами и временем, лежат текущие системные дата и время, установленные на вашем компьютере.

Со временем дела обстоят не так просто, как кажется на первый взгляд. Вспомните, что существуют временные зоны (часовые пояса), а также сезонные поправки времени. Так, например, текущее время в Санкт-Петербурге отличается от времени в Иркутске на 5 часов. Если в Иркутске уже полночь, то в Петербурге еще только 19 часов. Чтобы иметь возможность координировать деятельность во времени организаций и физических лиц в различных точках нашей планеты, была введена система отсчета времени. Она связана с меридианом, проходящим через астрономическую обсерваторию в городе Гринвич в Великобритании. Эту временную зону называют средним временем по Гринвичу (Greenwich Mean Time – GMT). Недавно кроме аббревиатуры GMT стали использовать еще одну – UTC (Universal Time Coordinated – Всеобщее Скоординированное Время).

Если системные часы вашего компьютера установлены правильно, то отсчет времени производится в системе GMT. Однако на Панели управления обычно устанавливается локальное время, соответствующее вашему часовому поясу. При создании и изменении файлов на вашем компьютере фиксируется именно локальное время. Вместе с тем операционная система знает разницу между локальным временем и GMT. При перемещении компьютера из одного часового пояса в другой необходимо изменить установки именно часового пояса, а не текущего системного времени (показания системных часов). Даты и время, генерируемые в сценариях, сохраняются в памяти в системе GMT, но пользователю выводятся, как правило, в локальном виде.

В программе на JavaScript нельзя просто написать 30.10.2002, чтобы получить значение даты, с которым в дальнейшем можно производить некие операции. Значения даты и времени создаются как экземпляры специального объекта Date. При этом объект сам будет «знать», что не бывает 31 июня и 30 февраля, а в високосных годах 366 дней.

Создание объекта даты. Объект даты создается с помощью выражения вида:

имяОбъектаДаты = new Date([параметры])

Параметры не обязательны, на что указывают квадратные скобки. Обратите внимание, что имяОбъектаДаты является объектом даты, а не значением какого-нибудь другого типа (например, строкой или числом).

Для манипуляций с объектом даты применяется множество методов объекта Date. При этом используется такой синтаксис:

переменная = имяОбъектаДаты.метод()

Если, например, объекту даты требуется присвоить новое значение, то для этого используется соответствующий метод

переменная = имяОбъектаДаты.метод(новое_значение)

Рассмотрим в качестве примера изменение значения года текущей системной даты:

```
xdate = new /* создание объекта, содержащего текущую дату и время */
Year = xdate.getYear()          /* в переменной Year содержится значение текущего года */
Year = Year +3 /* в переменной Year содержится значение, большее, чем текущий год,  на 3 */
xdate.setYear(Year)          /* в объекте устанавливается новое значение года */
```

При создании объекта даты с помощью выражения new Date() можно указать в качестве параметров, какие дату и время следует установить в этом объекте. Это можно сделать пятью способами:

```
new Date("месяц, дд, гггг чч:мм:сс")
new Date.e("Месяц дд, гггг")
new Date(гг, мм, дд, чч, мм, сс)
new Date(гг, мм, дд)
new Date(миллисекунды)
```

В первых двух способах параметры задаются в виде строки, в которой указаны компоненты даты и времени. Буквенные обозначения определяют шаблон параметров. Обратите внимание

на разделители – запятые и двоеточия. Время указывать не обязательно. Если компоненты времени опущены, то устанавливается значение 0 (полночь). Компоненты даты обязательно должны быть указаны. Месяц указывается в виде полного его английского названия (аббревиатуры не допускаются). Остальные компоненты указываются в виде чисел. Если число меньше 10, то можно писать одну цифру, не записывая ведущий 0 (например, 3:05:32).

В третьем и четвертом способах компоненты даты и времени представляются целыми числами, разделенными запятыми.

В последнем способе дата и время задаются целым числом, которое представляет количество миллисекунд, прошедших с начала 1 января года (то есть с момента 00:00:00). Количество миллисекунд, отсчитанное от указанной стартовой даты, позволяет вычислить все компоненты и даты, и времени.

Методы объекта Date. Для чтения и изменения информации о дате и времени, хранящейся в объекте даты, служат методы объекта Date. Напомним, что объект даты создается с помощью выражения:

имяОбъектаДаты = new Date([параметры])

Затем, чтобы применить метод метод() к объекту даты имяОбъектаДаты, следует написать:

имяОбъектаДаты.метод([параметры])

Довольно большое множество всех методов можно разделить на две категории: методы получения значений (их названия имеют префикс get) и методы установки новых значений (их названия имеют префикс set). В каждой категории выделяются две группы методов – для локального формата и формата UTC. Методы позволяют работать с отдельными компонентами даты и времени (годом, месяцем, числом, днем недели, часами, минутами, секундами и миллисекундами).

2.1.6. Объект Boolean (Логический)

Объект Boolean создается с помощью выражения вида:

переменная = new Boolean(логическое_значение)

Он имеет свойство prototype, методы to string() и значение of() которые имеют также объекты String и Number.

Смысл свойства prototype мы уже рассматривали применительно к объектам String и Array. Объект Boolean может понадобиться в том случае, когда всем логическим объектам, создаваемым с помощью выражения с ключевыми словами new Boolean, нужно добавить новые свойства или методы с помощью прототипа (свойства prototype).

2.1.7. Объект Function (Функция)

Создание объекта Function. Выше мы уже рассматривали стандартный способ определения функции:

```
function имя_функции(параметры)  {
код
}
```

Существует и другой способ, основанный на выражении с ключевыми словами new Function. Согласно этому способу функция создается как экземпляр объекта Function:

имя_функции = new Function(["пар1", ["парN"], "оператор1; [; операторN]")

Названия всех параметров являются строковыми значениями. Они разделяются запятыми. Заключительная строка содержит операторы кода тела функции, разделенные точкой с запятой.

Вызов функции, определенной как экземпляр объекта Function, можно выполнить обычным способом: имя_функции(параметры).

Примеры

```
Srectangle = new Function("width", "height", "var s =
width*height;
return  s")
Srectangle(2, 3) // возвращает 6
var expr = "var s = width*height;  return s" Srectangle =
new Function("width", "height", expr)
Srectangle(2, 3) // возвращает 6
a = "width" b = "height"
expr = "var s = width*height; return s" Srectangle = new
Function(a, b, expr)
Srectangle(2 , 3) // возвращает 6
```

При любом задании функции, стандартном или с помощью ключевого слова new, автоматически создается экземпляр объекта Function, который обладает своими свойствами и методами.

Свойства Function.

1. arguments – массив значений параметров, переданных функции.

Индексация элементов массива производится с 0. Поскольку это массив, он имеет свойства и методы объекта Array (в частности, свойство length – длина массива).

Свойство arguments применяется в теле определения функции, когда требуется проанализировать параметры, переданные ей при вызове. Например, можно узнать, сколько в действительности было передано параметров, не являются ли их значения пустыми (0, null) и т. п. Это свойство особенно полезно при разработке универсальных библиотечных функций.

Синтаксис выражения следующий: *имя_функции. arguments*

Пример

Функция, приводимая в этом примере, возвращает строку, содержащую значения параметров и их общее количество, которые были указаны в вызове функции (а не в ее определении!).

```
function myfunc(a, b,c){
var arglenth=myfunc.arguments.length    /* количество переданных параметров */
var x=""
for  (1=8; i< myfunc.arguments.length;i++)  {
x += myfunc.arguments [i] + "," }
return x+"Всего: "+ myfunc.arguments.length )
myfunc(5, "Вася") // "5,  Вася,  Всего:  2"
myfunc() // "Всего: 0"
myfunc(null,"",0,25) // "null , ,0,25, всего: 4"
```

2. length – количество параметров, указанных в определении функции. Синтаксис: *имя_функции. length*

В отличие от свойства arguments, количество параметров функции можно определить в программе за пределами тела этой функции.

Пример

```
function myfunc(a, b, c, d){ returnmyfunc.arguments.length
}
myfunc(a,b) //  2
myfunc.length // 4
```

3. caller – содержит ссылку на функцию, из которой была вызвана данная функция; если функция не вызывалась из другой функции, то значение этого свойства равно null.

Совместимость: IE4+, NN4.

Синтаксис: *имя_функции. caller*

В свойстве содержится все определение функции, из которой была вызвана функция имя.функции.

Методы Function. toString() – возвращает определение функции в виде строки. Синтаксис: *имя_функции.toString()*

Иногда этот метод используют в процессе отладки программ с помощью диалоговых окон.

2.1.8. Объект Object

Object является корневым объектом, на котором базируются все остальные объекты JavaScript, такие как String, Array, Date и т. д. В программах можно создавать свои собственные объекты. Это можно сделать различными способами.

Способ 1

```
function имя_конструктора ([napl,...[, napN]){
код
}
имяОбъекта = new имя_конструктора( ["nap1",...[, "napN"])
```

Способ 2

```
имяОбъекта = new Object()
имяОбъекта.свойство = значение
```

Способ 3

```
имяОбъекта = {свойство1: значение1  [, свойство2:  значение [...N.]}
```

Для обращения к свойствам и методам объекта используется следующий синтаксис:

```
ссылка_на_объект.свойство ссылка_на_объект.метод([параметры])
```

Допустим, например, что нам требуется создать объект Сотрудник, который содержал бы сведения о сотрудниках некоторой фирмы, такие как Имя, Отдел, Телефон, Зарплата и т. п. В фирме может быть много сотрудников, но сведения о них представляются в некоторой единой структуре. Эту структуру можно создать с помощью конструктора объекта:

```
function Сотрудник(Имя, Отдел, Телефон, Зарплата) {
this. Имя = Имя
this.Отдел = Отдел
this.Телефон = Телефон
this.Зарплата = Зарплата
}
```

Как видите, конструктор объекта может быть представлен в виде определения функции. Ключевое свойство this представляет ссылку на текущий, то есть определяемый конструктором объект. Все операторы присвоения с this, расположенные в теле функции-конструктора, определяют свойства объекта. В круглых скобках у имени объекта могут перечисляться параметры, чтобы иметь возможность создать конкретный объект, являющийся экземпляром обезличенного объекта Сотрудник, и присвоить его свойствам конкретные значения. Например, создадим конкретного сотрудника – agent007:

```
agent007 = new Сотрудник("Джеймс Бонд", 5, "223-332", 3600.50)
```

Доступ к свойствам этого объекта производится обычным способом:

```
agent007.Имя // "Джеймс Бонд"
agent007.Зарплата // 3600.5
```

Информация о новом сотруднике добавляется аналогичным образом:

```
Shtirlitz = new Сотрудник("Максимов", 4, "123-4567", 4500.50)
```

К свойствам и методам этого объекта обращаются довольно редко, и здесь мы не будем их подробно рассматривать. Отметим свойство prototype, назначение которого мы уже описывали применительно к объектам String и Array. Метод toString() обычно используется при отладке программ с помощью диалоговых окон.

Свойство hasOwnPropery("свойство") используется, чтобы определить, имеется ли у экземпляра объекта указанное свойство, определенное в его конструкторе (но не с помощью prototype). Если да, то возвращается true, в противном случае – false.

Более подробные сведения о создании объектов вы найдете в следующем параграфе.

Там приведено много примеров, и вы легко убедитесь, что понятие объекта не сложнее, чем понятие функции.

§ 2.2. Пользовательские объекты

Выше мы рассмотрели встроенные объекты, то есть заранее предопределенные в JavaScript и часто используемые в программах. С помощью выражений с ключевым словом new можно создавать экземпляры этих объектов, то есть их конкретные воплощения. Более того, благодаря свойству prototype имеется возможность добавлять к объектам новые свойства и методы, придуманные пользователем и отсутствовавшие в исходных встроенных объектах. В большинстве случаев, в частности при создании сценариев для веб-страниц, всего этого более чем достаточно. Однако нельзя обойти вниманием возможность создания собственных объектов. Зачем нужны собственные объекты?

Они не являются необходимыми для решения практических задач. С точки зрения программиста, объект представляет собой просто удобное средство организации данных и функций их обработки. Чтобы как-то организовать данные и функции в программе, далеко не всегда необходимо прибегать к такой конструкции, как объект. Ведь даже при внушительном количестве данных программист не всегда организует их в виде массива (объекта Array): бывает достаточным ограничиться простыми переменными.

В этом параграфе мы сосредоточимся на том, как создавать объекты, если возникла такая необходимость. Возможно, это вам понадобится. Кроме того, данный материал поможет лучше понять природу встроенных объектов, а также общую философию объектно-ориентированного программирования. Сейчас концепция объектно-ориентированного программирования является ведущей в технологиях создания крупных приложений, поэтому будет полезно познакомиться с ней поближе.

2.2.1. Создание объекта

Объекты в JavaScript можно создать несколькими способами. Мы рассмотрим три из них.

Первый способ основан на функции, в теле которой описываются все свойства и методы создаваемого объекта. Поскольку эта функция играет определяющую роль в создании объекта, ее называют функцией-конструктором или просто конструктором объекта. Что, собственно, должен делать конструктор? Очевидно, он должен ввести имя создаваемого объекта, а также его свойства и методы. Кроме того, он должен допускать возможность присвоения начальных значений свойствам.

Имя функции-конструктора объекта является одновременно и именем создаваемого объекта. Свойства и методы создаваемого объекта задаются в теле функции-конструктора с помощью операторов присвоения. При этом имена переменных-свойств записываются с ключевым словом this (этот).

Рассмотрим в качестве примера функцию-конструктор, определяющую объект car (автомобиль) со свойствами name (название), model (модель) и color (цвет):

```
function car(name, model, color)   {
 this, name = name
this.model = model
this.color = color
}
```

Эта функция определяет объект car. Чтобы создать конкретный экземпляр объекта следует выполнить выражение с вызовом этой функции, которой можно передать значения параметров. Мы уже знаем, что экземпляр объекта создается с помощью оператора присвоения с ключевым словом new:

```
mycar = new car("Ока",  "BA31111", "white")
```

Итак, мы создали объект mycar, являющийся экземпляром объекта car. Таких экземпляров с различными именами можно создать несколько. Значения свойств объекта mycar можно изменять в программе:

```
mycar.model // значение равно "BA31111"
mycar.name // значение равно "Ока"
mycar.model = "BA311113"      // значение равно "BA3 11113"
```

Как видите, объект – это просто особый способ группировки данных и их использования (составное имя переменной: объект.свойство).

Объекты можно создавать и с помощью конструктора new Object

mycar = new Object () mycar.name = "Ока" mycar.model = "BA31111" mycar.color = "white"

В этом способе отчетливо видно: создаваемый объект является экземпляром объекта Object, подобно тому, как, например, создаваемый массив является экземпляром объекта Array.

Допускается также и следующая компактная запись определения объекта:

mycar = {name : "Ока" , model:"ВАЗ 1111" , color:"white")

Здесь все определение свойств заключается в фигурные скобки. Пары *свойство = значение* разделяются запятыми, а имя свойства отделяется от значения двоеточием.

При создании объекта мы можем задать значения свойств по умолчанию, то есть значения, которые будут иметь свойства, если при создании экземпляра этого объекта значения его свойств не указаны явным образом (то есть имеют значения null, 0 или ""). Это делается с помощью логического оператора ИЛИ (обозначаемого ||) в конструкторе объекта в виде функции:

```
function car(name, model, color) { // конструктор объекта car
this.name = name || "неизвестно"
this.model = model || "неизвестно"
this.color = color || "black"
mycar = new car("Жигули', "")  /* создание экземпляра mycar объекта car */
mycar.name // "Жигули"
mycar.model // "неизвестно"
mycar.color // "black"
```

2.2.2. Добавление свойств

Если возникает необходимость добавить новое свойство к существующему объекту, а также ко всем его экземплярам, то это можно сделать с помощью свойства prototype. В приведенном ниже примере мы создаем объект car (автомобиль), затем его экземпляр

mycar, а затем добавляем к свойству prototype свойство owner (владелец) с конкретным значением:

```
function car(name, model,  color)  {     // конструктор объекта car
this.name = name  ||  "неизвестно."
this.model = model  ||  "неизвестно"
this.color = color  ||  "black"
}
mycar = new car("Жигули","") /* создание экземпляра mycar объекта car */
mycar.name //  "Жигули"
mycar.model //  "неизвестно"
mycar.color //  "black"
car.prototype.owner = "Иванов" // добавляем новое свойство
mycar.owner //  "Иванов"
```

Если нужно добавить новое свойство только к конкретному объекту (к данному экземпляру объекта), то это можно сделать просто с помощью оператора присваивания:

```
имяОбъекта.новое_свойство = значение
```

В следующем примере мы создаем объект вообще без свойств, а затем добавляем его экземпляры, и к ним – различные свойства.

```
function car(){}
mycarl = new car()
mycar2 = new car()
mycar1.name =  "Жигули"
mycar2.model =  "BA32106"
mycar1.name //  " Жигули "
mycar1.model //  undefined  (не определено)
mycar2.name // undefined  (не определено)
mycar2.model // "BA32106"
```

2.2.3. Связанные объекты

В объекте в виде свойства может содержаться ссылка на другой объект. В этом случае оба объекта оказываются связанными: один из них оказывается подобъектом или, другими словами, свойством другого. Например, мы можем создать объект photo, содержащий в качестве своих свойств название автомобиля

и URL-адрес файла с его изображением. Такой объект можно связать с объектом car, содержащим основную информацию об автомобилях, добавив к нему свойство, значением которого является ссылка на объект photo.

§ 2.3. Специальные операторы

В этом параграфе описываются операторы, которые при программировании на JavaScript используются относительно редко. Возможно, они вам когда-нибудь потребуются.

2.3.1. Побитовые операторы

Побитовые (поразрядные) операторы применяются к целочисленным значениям и возвращают целочисленные значения. При их выполнении операнды предварительно приводятся к двоичной форме представления, в которой число является последовательностью из нулей и единиц длиной 32. Эти нули и единицы называются двоичными разрядами, или битами. Далее производится некоторое действие над битами, в результате которого получается новая последовательность битов. В конце концов эта последовательность битов преобразуется к обычному целому числу – результату побитового оператора. В табл. 2.2 приведен список побитовых операторов.

Таблица 2.2

Побитовые операторы

Оператор	Название	Левый операнд	Правый операнд
&	Побитовое «и»	Целое число	Целое число
|	Побитовое«или»	Целое число	Целое число
^	Побитовое исключающее «или»	Целое число	Целое число
~	Побитовое «не»	—	Целое число
<<	Смещение влево	Целое число	Количество битов, на которое производится смещение
>>	Смещение вправо	Целое число	Количество битов, на которое производится смещение
>>	Заполнение нулями при смещении вправо	Целое число	Количество битов, на которое производится смещение

Операторы &, | и ~ напоминают логические операторы, но их область действия – биты, а не логические значения. Оператор ~ изменяет значение бита на противоположное: 0 на 1, а 1 – на 0. В табл. 2.3 поясняется, как работают операторы &, |, ^.

Таблица 2.3

Работа операторов &

X	*Y*	*X*&*Y*	*X*|*Y*	*X*^*Y*
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

Например, 2&3 равно 2, а 2|3 равно 3. Действительно, в двоичном представлении 2 есть 10, а 3 – 11. Применение побитовых операторов даст в случае оператора & двоичное число 10, то есть десятеричное число 2, а в случае оператора | – двоичное число 11, то есть десятеричное 3.

У операторов смещения один операнд и один параметр, указывающий, на какое количество битов следует произвести смещение. Например, 3<<2 равно 12, потому что смещение влево на два бита двоичного числа 11 (десятичное 3) дает 1100, что в десятичной форме есть 12. Результат вычисления выражения 6>>2 – 1. Действительно, число 6 в двоичной форме – это 110; смещение его вправо на два бита дает 1 как в двоичной, так и в десятичной форме.

2.3.2. Объектные операторы

Оператор удаления свойств объекта (delete). Удалить свойство объекта, а также элемент массива можно с помощью оператора delete. Обычно этот оператор используют для удаления элементов массива: delete элемент

ВНИМАНИЕ! При удалении элемента массива удаляется и его индекс, но оставшиеся элементы сохраняют свои прежние индексы, а длина массива не изменяется.

Пример

```
myarray = new Array("a" , "b" , "c" , "d")
myarray.length // 4
```

```
delete myarray[1]
myarray.length // 4
myarray[0] // "a"
myarray[1] // undefined
myarray[2] // "c"
myarray[3] // "d"
```

Использование оператора delete не приводит к немедленному освобождению памяти. Решение об освобождении памяти принимается так называемым ядром JavaScript, а пользовательская программа лишь создает к этому предпосылки, но не может контролировать этот процесс абсолютно.

Оператор проверки наличия свойств (in). Этот оператор позволяет проверить, имеется ли некоторое свойство или метод у того или иного объекта. Левый операнд представляет собой ссылку в виде строки на интересующее нас свойство или метод, а правый операнд – объект. Ссылка на метод содержит лишь его название без круглых скобок. Если свойство или метод содержится в объекте, то возвращается true, иначе – false. Отсюда следует, что оператор in можно применять в условных выражениях (в операторах if, switch, for, while, do-while).

Оператор проверки принадлежности объекта модели (instanceof). Этот оператор позволяет проверить, принадлежит ли некоторый объект объектной модели JavaScript. Левый операнд представляет проверяемое значение, а правый – ссылку на корневой объект, такой как Array, String, Date и т. п. Выражение с оператором instanceof возвращает true или false и, таким образом, может использоваться в условных выражениях (в операторах if, switch, for, while, do-while).

Оператор in поддерживается браузерами IE5.5+, NN6+.

Пример

Созданный массив является экземпляром объекта Array, а последний сам является экземпляром корневого объекта Object.

```
myarray = new Array()
myarray instanceof Array // true
Array instance Object // true
Myarray instanceof String // false
```

2.3.3. Комплексные операторы

Оператор условия (?:). Этот оператор является сокращенной формой оператора условного перехода if… else... Его так и называют: оператор условия. Обычно он используется вместе с оператором присвоения одного из двух возможных значений, в зависимости от значения условного выражения. Синтаксис оператора условия следующий:

условие ? выражение1 : выражение2

С оператором присвоения оператор условия имеет такой вид:

переменная = условие ? выражение1 : выражение2

Оператор условия возвращает значение выражения выражение1, если условие истинно, в противном случае – значение выражения выражение2

Пример

```
d = new Date()
x = d.getDate()
typedate = (x%2 == 0)&&(x > 1) ? "четное" : "нечетное"
```

Если бы в JavaScript не было оператора условия, то пришлось бы написать функцию, которая работала бы как оператор условного перехода if, но, в отличие от него, возвращала значение. Вот пример этой функции:

```
function if(condition, expr1, expr2){ if (condition)
return eval(expr1)
else
return eval(expr2)
}
```

Для данной функции нельзя было выбрать название if, поскольку это ключевое слово. Потому было взято наиболее близкое к нему. Обратите внимание: предполагается, что выражения передаются функции как строки, содержащие выражения. Однако допустимы значения и других типов.

Оператор определения типа (typeof). Этот оператор используется для проверки, относится ли значение к одному из следующих типов: string, number, boolean, object, function или undefined.

Значение, возвращаемое оператором typeof, является строковым. Оно содержит одно из перечисленных выше названий типа. Единственный операнд пишется справа от ключевого слова typeof.

§ 2.4. Приоритеты операторов

Выше мы уже говорили, что в выражениях операторы выполняются в порядке, определяемом с помощью круглых скобок, согласно приоритетам; операторы с одинаковыми приоритетами выполняются слева направо. В этом параграфе мы уточним приоритеты всех операторов (табл. 2.4).

Круглые скобки, с помощью которых можно установить любой порядок выполнения операторов в выражении, можно считать операторами. Они обладают наивысшим приоритетом. Эти скобки могут образовывать структуру вложенности. Выражения, заключенные во внутренние круглые скобки, выполняются раньше тех, которые заключены во внешние круглые скобки. Интерпретатор JavaScript начинает анализ выражения именно с выяснения структуры вложенности пар круглых скобок (табл. 2.4).

Таблица 2.4

Распределение операторов по приоритетам

Приоритет	Оператор	Комментарий
1	( )	От внутренних к внешним
	[ ]	Значение индекса массива
	function()	Вызов функции
2	!	Логическое «не»
	~	Побитовое «не»
	-	Отрицание
	++	Инкремент (приращение)
	--	Декремент
	new	
	typeof	
	void	
	delete	Удаление объектного элемента
3	*	Умножение
	/	Деление
	%	Деление по модулю (остаток от деления)

Окончание табл. 2.4

Приоритет	Оператор	Комментарий
4	+	Сложение (конкатенация)
	-	Вычитание
5	<<	Побитовые сдвиги
	>>	
	>>>	
6	<	Меньше
	<=	Не больше (меньше или равно)
	>	Больше
	>=	Не меньше (больше или равно)
7	==	Равенство
	!=	Неравенство
8	&	Побитовое «и»
9	^	Побитовое исключающее «или»
10	|	Побитовое «или» (дизъюнкция)
11	&&	Логическое «и» (конъюнкция)
12	||	Логическое «или»

Кроме приоритетов следует также учитывать, что сложные логические выражения, состоящие из нескольких более простых, соединенных операторами И и ИЛИ, выполняются по так называемому принципу короткой обработки. Это означает, что значение всего выражения бывает можно определить, вычислив лишь одно или несколько более простых выражений, не вычисляя остальные. Например, выражение *x&&y* вычисляется слева направо; если значение *x* оказалось равным false, то значение *y* не вычисляется, поскольку и так известно, что значение всего выражения равно false. Аналогично, если в выражении *x* || *y* значение *x* равно true, то значение *y* не вычисляется, поскольку уже ясно, что все выражение равно true.

§ 2.5. Зарезервированные ключевые слова

Ключевые слова JavaScript, применяемые для обозначения элементов синтаксиса языка, а также другие, зарезервированные на будущее, нельзя использовать в качестве имен переменных, функций и объектов. В крайнем случае можете просто несколько модифицировать ключевое слово, изменив регистр, добавив какой-нибудь префикс (например, символ подчеркивания) или суффикс.

ВНИМАНИЕ! Использование символов в различном регистре часто приводит к недоразумениям, обусловленным тем, что одна и та же по вашему замыслу переменная из-за небрежности встречается в программе в различном регистре. Поскольку JavaScript является регистрозависимым языком, он воспринимает эти переменные как различные.

Если вы хотите использовать различный регистр в именах, то придерживайтесь определенных правил. Например, только первый символ имени всегда является прописной буквой.

Практические задания к главе 1

Примечание

При воспроизведении кода могут возникнуть проблемы с определением кодировки, на которой написана страница. Вместо ожидаемого "Привет" можно получить нечто вроде "аааНод". Потому следует в начале каждой страницы для браузера указывать кодировку каждой страницы. Сделать это можно с помощью простого тега <meta charset="UTF-8" />. <meta charset="" /> – непосредственно сам скрипт. А данные в "" – нужная вам кодировка. В данном случае используется кодировка Юникод.

Задание 1

Написать простейший скрипт для выведения альтернативных календарных данных с применением массива, стандартного объекта Date и функции работы с ним.

Возможное решение

```
<script type="text/javascript" language="JavaScript">
<!--
var day = new Array ("Воскреснье","Понедельник","Вторник",
"Среда","Четверг","Пятница","Суббота");
var month = new Array ("Утренней Звезды", "Восхода", "Первоцвета", "Дождя", "Сева", "Середины Года", "Солнцеворота",
"Урожая", "Огня", "Мороза", "Заката", "Вечерней Звезды");
tdate = new Date();
```

```
tday = tdate.getDay();
tmonth = tdate.getMonth();
tweekday= tdate.getDate();

document.write("<font size=\"3\" color=\"#fbb52e\">" + day[tday] +
", " + "<font color=\"#f90b0b\">" + tweekday + "</font>" + " месяц " +
month[tmonth] + ".</font>");
-->
</script>
```

Где функции:

getDay возвращает целое число, обозначающее день недели: 0 – воскресенье, 1 – понедельник, 2 – вторник, и т. д.

getMonth возвращает целое число от 0 до 11. 0 соответствует январю, 1 – февралю, и т. д.

getDate Возвращает целое число от 1 до 31.

Задание 2

Как правило, если добавляется на сайте календарь, то есть и собственные часы, которые будут отображать текущее время. Удобство пользователя всегда ставится на первое место при разработке сайта.

Возможное решение

```
  <script type="text/javascript">
 function digitalWatch() {
   var date = new Date();
   var hours = date.getHours();
   var minutes = date.getMinutes();
   var seconds = date.getSeconds();
   if (hours < 10) hours = "0" + hours;
   if (minutes < 10) minutes = "0" + minutes;
   if (seconds < 10) seconds = "0" + seconds;
document.getElementById("digital_watch").innerHTML = hours + ":"
+minutes + ":" + seconds;
   setTimeout("digitalWatch()", 1000);
 }
</script>
```

Здесь мы используем скрипт, который создает функцию "цифровые часы", где используется уже упомянутый ранее стандартный объект Date и используются функции, только возвращают они уже не дату, а время.

Код добавляется в html документ между тегами <head> и </head>.

Вызов функций располагается в нужном месте между тегами <body> и </body>.

<body bgcolor="#141414" onload="digitalWatch()"> – прописываем как атрибут тега body

* * – вызов функции

Задание 3

Написать две функции, которые размещают два изображения. Одно будет все статично отображаться, но при наведении на него мыши – меняться на следующее изображение.

Возможное решение

```
<SCRIPT type="text/javascript">
  function swap1for2()
    {  document.pic1.src="bubble2.gif" }
  function swap2for1()
    { document.pic1.src="bubble1.gif" }
 </SCRIPT>
```

Этот код добавляется между тегами <head> и </head>.

```
<a href="" onMouseOver="swap1for2() "
   onMouseOut="swap2for1()">
 <img src="bubble1.gif" border=0 name="pic1"> </a>
```

Этот код добавляется между тегами <body> и </body>.

Задание 4

Написать скрипт, уведомляющий вас о выборе варианта фона сайта из предложенных

```
<SCRIPT type="text/javascript">
var user_name = prompt ("Можно узнать, как Вас зовут?","Ваше имя");
function newcolor(color)
{
alert(user_name + "! Вы выбрали " + color)
document.bgColor=color
}
</SCRIPT>
```

Этот код добавляется между тегами <head> и </head>.

```
<form>
  <input type="button" value="Оранжевый" onClick="new-
color('orange')">
  <input type="button" value="Салатовый" onClick="new-
color('lightgreen')">
</form>
```

Этот код добавляется между тегами <body> и </body>.

Собранные вместе элементы:

```
<meta charset="UTF-8" />
<head>

<title>Пример </title>
<SCRIPT type="text/javascript">
var user_name = prompt ("Можно узнать, как Вас зовут?","Ваше
имя");
function newcolor(color)
{
alert(user_name + "! Вы выбрали " + color)
document.bgColor=color
}

</SCRIPT>
 <center>

   <SCRIPT type="text/javascript">
   function swap1for2()
     {  document.pic1.src="cropped-57901.jpg" }
   function swap2for1()
     { document.pic1.src="cropped-1269173005_2008simonhawk_038.jpg" }
 </SCRIPT>
    <script type="text/javascript">
 function digitalWatch() {
   var date = new Date();
   var hours = date.getHours();
   var minutes = date.getMinutes();
   var seconds = date.getSeconds();
```

```
        if (hours < 10) hours = "0" + hours;
        if (minutes < 10) minutes = "0" + minutes;
        if (seconds < 10) seconds = "0" + seconds;
      document.getElementById("digital_watch").innerHTML = hours
+ ":" +minutes + ":" + seconds;
        setTimeout("digitalWatch()", 1000);
       }
      </script>
          </center>
      </head>
      <body bgcolor="#141414" onload="digitalWatch()">

      <div class="content_center" align="center">
      <script type="text/javascript" language="JavaScript">
      <!--
      var day = new Array ("Воскресенье","Понедельник","Вторник",
"Среда","Четверг","Пятница","Суббота");
      var month = new Array ("Утренней Звезды", "Восхода", "Пер-
воцвета", "Дождя", "Сева", "Середины Года", "Солнцеворота",
"Урожая", "Огня", "Мороза", "Заката", "Вечерней Звезды");

      tdate = new Date();
      tday = tdate.getDay();
      tmonth = tdate.getMonth();
      tweekday= tdate.getDate();
      document.write("<font size=\"3\" color=\"#fbb52e\">" +
day[tday] + ", " + "<font color=\"#f90b0b\">" + tweekday +
"</font>" + " месяц " + month[tmonth] + ".</font>");
      -->

      </script> <p id="digital_watch" style="color: #f00; font-size:
120%; font-weight: bold;"></p>
       <a href="" onMouseOver="swap1for2() "
         onMouseOut="swap2for1()">
       <img src="cropped-57901.jpg" border=0 name="pic1"> </a>
      </div>
      </br>
      </br>
      </br>
      </br>
```

```
</br>
</br>
</br>
</br>
</br>
</br>
</br>
</br>
</br>
<center>
 <hr width=100%>
 Выберите дизайн, который вам больше нравится:
 <form>
  <input type="button" value="Оранжевый" onClick="new-
color('orange')">
  <input type="button" value="Салатовый" onClick="new-
color('lightgreen')">
 </form>
 </center>
</body>
</html>
```

Результат выполнения кода представлен на рис. 2.1.

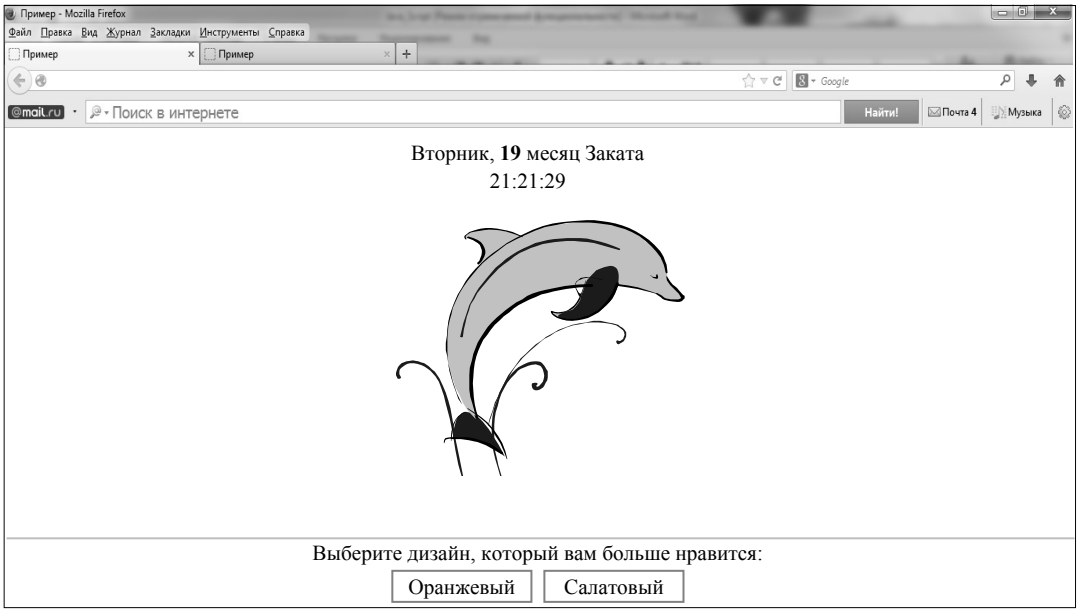

Рис. 2.1. Пример выполнения кода

Практические задания к главе 2

Задание 1

Написать скрипт, показывающий количество оставшегося времени до установленной даты. Допустим, что нужная дата – Новый год.

Предлагаемый вариант

```
<Script Language="JavaScript">
function myFunction1(){
if (date<=myDate) month1=12-(month-myMonth);
else month1=11-(month-myMonth);}
function myFunction2(){
if (date>myDate) month1=(myMonth-month-1);
else month1=(myMonth-month);}
function myFunction3(){
if (date<=myDate) month1=0;
else month1=11;}
function myFunction4(){
if (month==2) date1=(28-date+myDate);
if (month==4) date1=(30-date+myDate);
if (month==6) date1=(30-date+myDate);
if (month==9) date1=(30-date+myDate);
if (month==11) date1=(30-date+myDate);
if (month==1) date1=(31-date+myDate);
if (month==3) date1=(31-date+myDate);
if (month==5) date1=(31-date+myDate);
if (month==7) date1=(31-date+myDate);
if (month==8) date1=(31-date+myDate);
if (month==10) date1=(31-date+myDate);
if (month==12) date1=(31-date+myDate);}
var date=(new Date()).getDate();
var month=(new Date()).getMonth()+1;
var myDate=01
var myMonth=01
if (month>myMonth) myFunction1();
if (month<myMonth) myFunction2();
if (month==myMonth) myFunction3();
if (date<=myDate) date1=(myDate-date);
```

```
if (date>myDate) myFunction4();
if (month1==1) monthtxt=("месяц");
if (month1>1 && month1<5) monthtxt=("месяца");
if (month1>4) monthtxt=("месяцев");
if (date1==1, 21) datetxt=("день");
if (date1>1 && date1<5) datetxt=("дня");
if (date1>21 && date1<25) datetxt=("дня");
if (date1>4 && date1<21) datetxt=("дней");
if (date1>24 && date1<31) datetxt=("дней");
var txt=("До Нового года осталось:");
if (month1==0 && date1==0) txt=("Сегодня 1 января");
document.write(txt+' ');
if (month1>0) document.write(month1+" "+monthtxt+' ');
else document.write("");
if (date1>0) document.write(date1+" "+datetxt);
else document.write("");
</Script>
```

Задание 2

Написать функцию альтернативного выбора цвета фона.

Предлагаемый вариант

Добавляется в шапку документа:

```
<script language="javascript">
function go()
{selindex=document.form.select.selectedIndex
chosenvalue=document.form.select.options[selindex].value
document.bgColor=chosenvalue}
function change()
{chosentext=document.form.select.options[document.form.select.
selectedIndex].text}
</script>
```

Добавляется в тело документа:

```
<form name="form">
<select name="select" onchange="change()">
<option value="#ffffff">Белый
<option value="#ff0000">Красный
<option value="#FFA500">Оранжевый
```

```
<option value="#ffff00">Желтый
<option value="#008000">Зеленый
<option value="#ADD8E6">Голубой
<option value="#0000ff">Синий
<option value="#800080">Фиолетовый
</select>
<input type="button" name="btn" value="Изменить цвет" onclick="go()">
</form>
```

Задание 3

Написать функцию автоматического вывода последнего обновления страницы.

Предлагаемый код

```
<script language="JavaScript">
<!--
document.writeln(document.lastModified);
//-->
</script>
<BODY onUnload="ByeWin()" >
```

Собранные вместе элементы, включая все теги из первого практического задания:

```
<meta charset="UTF-8" />
<head>

<script language=JavaScript>
<!--
var message="М-да! Облом! Правый клик мышкой";

function clickIE4(){
if (event.button==2){
alert(message);
return false;
}
}
function clickNS4(e){
if (document.layers||document.getElementById&&!document.all){
if (e.which==2||e.which==3){
```

```
alert(message);
return false;
}
}
}
if (document.layers){
document.captureEvents(Event.MOUSEDOWN);
document.onmousedown=clickNS4;
}
else if (document.all&&!document.getElementById){
document.onmousedown=clickIE4;
}
document.oncontextmenu=new Function("alert(message);return false")
// -->
</script>

<script language="javascript">
function go()
{selindex=document.form.select.selectedIndex
chosenvalue=document.form.select.options[selindex].value
document.bgColor=chosenvalue}
function change()
{chosentext=document.form.select.options[document.form.select.
selectedIndex].text}
</script>

<title>Пример </title>
<SCRIPT type="text/javascript">
var user_name = prompt ("Можно узнать, как Вас зовут?","Ва-
ше имя");
function newcolor(color)
{
alert(user_name + "! Вы выбрали " + color)
document.bgColor=color
}

</SCRIPT>
 <center>

   <SCRIPT type="text/javascript">
```

```
    function swap1for2()
      { document.pic1.src="cropped-57901.jpg" }
    function swap2for1()
     { document.pic1.src="cropped-1269173005_2008simonhawk_038.jpg" }
  </SCRIPT>

    <script type="text/javascript">
  function digitalWatch() {
    var date = new Date();
    var hours = date.getHours();
    var minutes = date.getMinutes();
    var seconds = date.getSeconds();
    if (hours < 10) hours = "0" + hours;
    if (minutes < 10) minutes = "0" + minutes;
    if (seconds < 10) seconds = "0" + seconds;
  document.getElementById("digital_watch").innerHTML = hours
+ ":" +minutes + ":" + seconds;
    setTimeout("digitalWatch()", 1000);
  }
  </script>
     </center>
  </head>

  <body bgcolor="#141414" onUnload="ByeWin()" onload="digital-
Watch()" onLoad="showtime()" text="white" link="#FFFF99"
alink="#FFFF99" vlink="#FFFF99">

   <div class="content_center" align="center">
  <script type="text/javascript" language="JavaScript">
  <!--
  var day = new Array ("Воскресенье","Понедельник","Вторник",
"Среда","Четверг","Пятница","Суббота");
  var month = new Array ("Утренней Звезды", "Восхода", "Пер-
воцвета", "Дождя", "Сева", "Середины Года", "Солнцеворота",
"Урожая", "Огня", "Мороза", "Заката", "Вечерней Звезды");

  tdate = new Date();
  tday = tdate.getDay();
  tmonth = tdate.getMonth();
```

```
tweekday= tdate.getDate();

document.write("<font size=\"3\" color=\"#fbb52e\">" + day[tday] + ", " + "<font color=\"#f90b0b\">" + tweekday + "</font>" + " месяц " + month[tmonth] + ".</font>");
-->

</script> <p id="digital_watch" style="color: #f00; font-size: 120%; font-weight: bold;"></p>

 <a href="" onMouseOver="swap1for2() "
   onMouseOut="swap2for1()">
 <img src="cropped-57901.jpg" border=0 name="pic1"> </a>
</br>
<Script Language="JavaScript">
function myFunction1(){
if (date<=myDate) month1=12-(month-myMonth);
else month1=11-(month-myMonth);}
function myFunction2(){
if (date>myDate) month1=(myMonth-month-1);
else month1=(myMonth-month);}
function myFunction3(){
if (date<=myDate) month1=0;
else month1=11;}
function myFunction4(){
if (month==2) date1=(28-date+myDate);
if (month==4) date1=(30-date+myDate);
if (month==6) date1=(30-date+myDate);
if (month==9) date1=(30-date+myDate);
if (month==11) date1=(30-date+myDate);
if (month==1) date1=(31-date+myDate);
if (month==3) date1=(31-date+myDate);
if (month==5) date1=(31-date+myDate);
if (month==7) date1=(31-date+myDate);
if (month==8) date1=(31-date+myDate);
if (month==10) date1=(31-date+myDate);
if (month==12) date1=(31-date+myDate);}
var date=(new Date()).getDate();
```

```
var month=(new Date()).getMonth()+1;
var myDate=01
var myMonth=01
if (month>myMonth) myFunction1();
if (month<myMonth) myFunction2();
if (month==myMonth) myFunction3();
if (date<=myDate) date1=(myDate-date);
if (date>myDate) myFunction4();
if (month1==1) monthtxt=("месяц");
if (month1>1 && month1<5) monthtxt=("месяца");
if (month1>4) monthtxt=("месяцев");
if (date1==1, 21) datetxt=("день");
if (date1>1 && date1<5) datetxt=("дня");
if (date1>21 && date1<25) datetxt=("дня");
if (date1>4 && date1<21) datetxt=("дней");
if (date1>24 && date1<31) datetxt=("дней");

var txt=("<font size=\"3\" color=\"#fbb52e\">" +"До Нового года осталось:");
if (month1==0 && date1==0) txt=("<font size=\"3\" color=\"#fbb52e\">"+ ", " +"Сегодня 1 января!");
document.write(txt+' ');
if (month1>0) document.write(month1+" "+monthtxt+' ');
else document.write("");
if (date1>0) document.write(date1+" "+datetxt);
else document.write("");
</Script>
 </div>
</br>
</br>
</br>
</br>
</br>
</br>
</br>
</br>
</br>
<center>
 <hr width=100%>
```

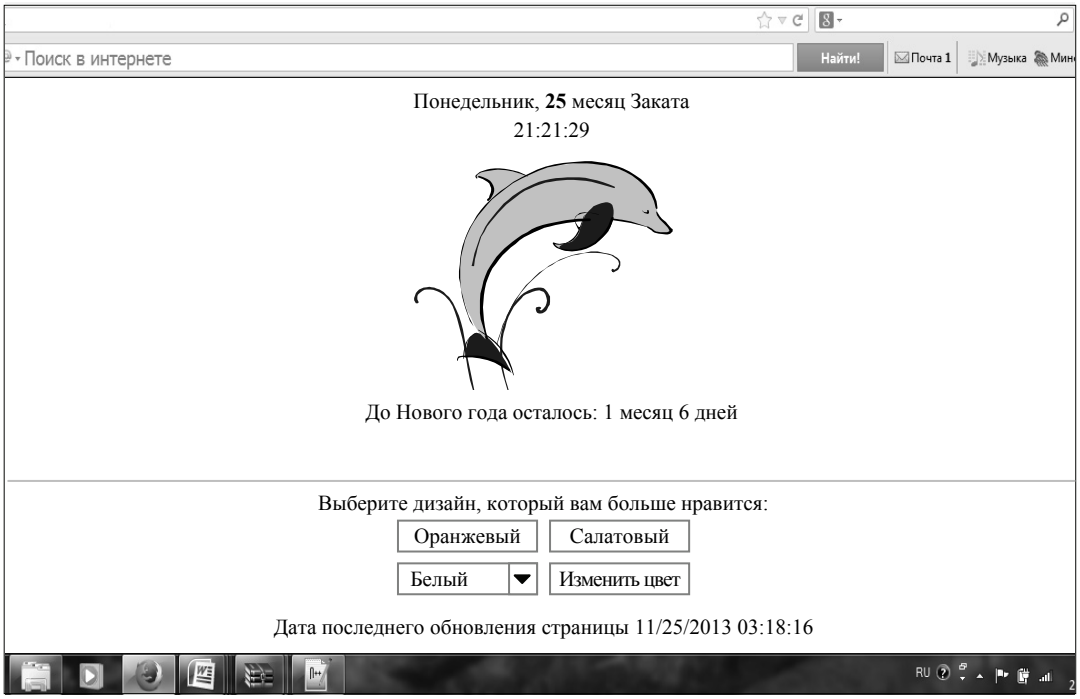

Рис. 2.2. Пример выполнения кода

Выберите дизайн, который вам больше нравится:

```
<form>
<input type="button" value="Оранжевый" onClick="newcolor('orange')">
<input type="button" value="Салатовый" onClick="newcolor('lightgreen')">
</form>
<form name="form">
<select name="select" onchange="change()">
<option value="#ffffff">Белый
<option value="#ff0000">Красный
<option value="#FFA500">Оранжевый
<option value="#ffff00">Желтый
<option value="#008000">Зеленый
<option value="#ADD8E6">Голубой
<option value="#0000ff">Синий
<option value="#800080">Фиолетовый
</select>
<input type="button" name="btn" value="Изменить цвет" onclick="go()">
```

```
</form> <script language="JavaScript">
<!--
 document.write("Дата последнего обновления страницы" +" "
+ document.lastModified);
//-->
</script></FONT color>
  </center>
</body>
</html>
```

Пример выполнения описанного кода представлен на рис. 2.2.

Задание 4

Создайте еще один файл html. В нем создать сценарий, который основан на использовании свойства tagName, которое возвращает название тега, с помощью которого был создан соответствующий объект.

```
<HTML>
<H1> Пример2</H1>
<IMG SRC = "Xw9wMLeCujs.jpg">
</br>
<p>
<FORM>
<INPUT TYPE = "text" VALUE = "">
<p>
<BUTTON onclick = "my()">Нажми здесь</BUTTON>
</FORM>
<SCRIPT>
msg = ""
for(i = 0; i < document.all . length; i++){
msg += i + " " + document . all [i] . tagName + "\n"
}
alert ( msg)
</SCRIPT>
</HTML>
```

В диалоговом окне, выведенном с помощью alert(), перечислены теги HTML-документа. Их порядковые номера соответствуют индексам в коллекции all (рис. 2.3).

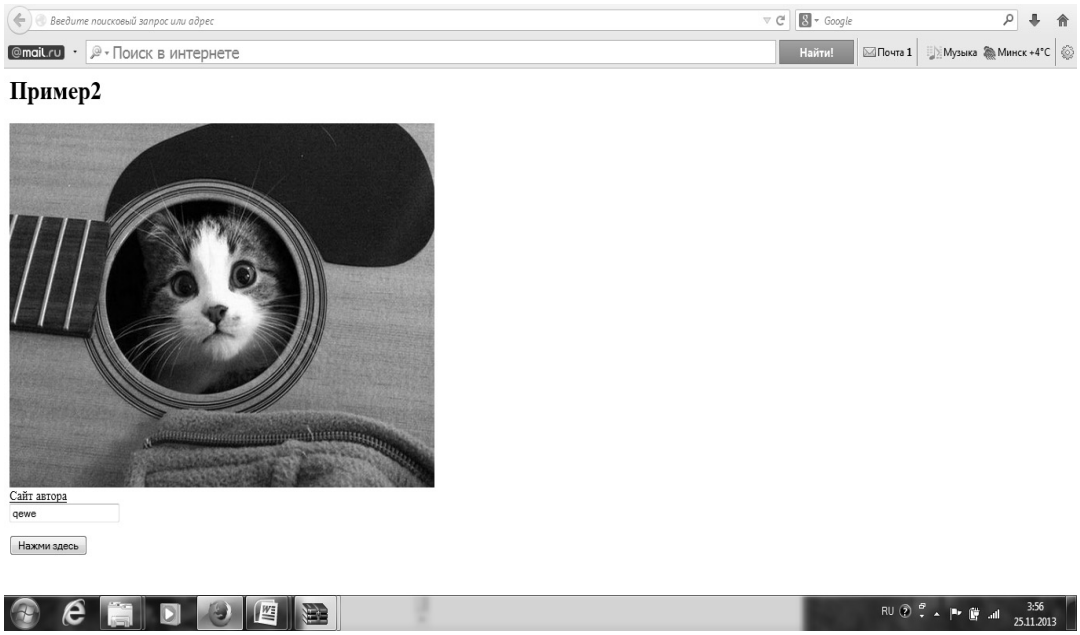

Рис. 2.3. Пример выполнения кода

Задание 5

Написать и добавить в конец файла скрипт, показывающий откуда пользователь перешел на вашу страницу.

```
<SCRIPT language=JavaScript>
var where = document.referrer;
if(where==0){document.write("Вы  загрузились с пустой страницы")}
else {document.write("Вы пришли с сайта "+where)}
</SCRIPT>
</center>
```

Глава 3
ОСНОВЫ СОЗДАНИЯ СЦЕНАРИЕВ

§ 3.1. Простой HTML

Для создания веб-страниц разработан довольно простой язык HTML (Hyper Text Markup Language – язык разметки гипертекста). Изначально он задумывался как язык разметки документа, содержащего текстовую и графическую информацию вместе с элементами управления, такими как ссылки на другие документы. Тексты, содержащие элементы, которые являются ссылками на другие документы, называются гипертекстами, а сами ссылки называют еще гиперссылками. Кроме текстовой и графической информации, а также ссылок в документ можно включать аудио- и видеофрагменты. Чтобы создать такой документ, достаточно в обычном текстовом редакторе (например, в Блокноте Windows) написать программу на языке HTML. Такие программы сохраняются в файлах с расширением htm или html, а выполняются они в веб-браузере, например Microsoft Internet Explorer или Netscape Navigator. Инструкции HTML называют тегами, или дескрипторами. Они выделяются угловыми скобками (< и >). Каждый тег имеет свое название, являющееся ключевым словом. Спецификация тега обеспечивается его параметрами, которые обычно называют атрибутами. Атрибуты могут иметь значения (аргументы). Названия тегов и атрибуты можно писать в любом регистре. Атрибуты, если они имеются, пишутся за названием тега в произвольном порядке через пробел. Формат тега имеет следующий вид:

<НАЗВАНИЕ_ТЕГА [АТРИБУТ1 [= значение1] . . . [АТРИБУТN [= значениеN]]>

Квадратные скобки показывают, что заключенные в них элементы не обязательны. Существуют теги без атрибутов (например, тег перехода на новую строку < *BR*>), а также атрибуты без аргументов (например, *NORESIZE* в теге <*FRAME*>). Даже если для

тега предусмотрены атрибуты, во многих случаях их можно не указывать, используя значение по умолчанию. Так, тег *<IMG SRC = "picture.jpg">* дает браузеру команду вывести на экран графический объект из файла picture.jpg. Здесь ключевое слово *IMG* – название тега, SRC – атрибут, а *"picture.jpg"* – аргумент (значение). Атрибут ID присущ любому тегу, но если он не используется, то по умолчанию принимается, что его значение – пустая строка. Теги можно писать в одной строке или в нескольких, делая переносы в любом месте. Невидимый служебный символ перевода строки и возврата каретки, вставляемый в редакторе при нажатии клавиши Enter, не воспринимается браузером как разделитель тегов. Можно считать, что браузер воспринимает последовательность тегов как единую строку символов.

Большинство тегов являются контейнерными. Это означает следующее.

• Тегу *<ТЕГ>* обязательно соответствует заключительный тег *</ТЕГ>*.

• Между этими тегами можно разместить другие теги, контейнерные или нет.

В связи с этим можно говорить о вложенности одних тегов в другие.

Например, теги тела документа *<BODY>*, ссылки *<A . . .>*, раздела *<DIV>*, таблицы *<TABLE>* и др. являются контейнерными, то есть им соответствуют заключительные теги *</BODY>, </A>, </DIV>*, *</TABLE>*. Говоря «внутри тега *<ТЕГ>*», мы имеем в виду то, что расположено между тегами *<ТЕГ>* и *</ТЕГ>*. В приведенном ниже примере показано, что внутри тега ссылки *<A>* можно разместить тег графического изображения *<IMG>*, создав таким образом графическую ссылку:

<A HREF = "www.yandex.ru" > < IMG SRC = " banner.gif " > < / A >

HTML-программа (HTML-код), формирующая документ, начинается тегом *< HTML>* и заканчивается тегом *</HTML>*. Таким образом, *<HTML>* является контейнерным тегом, причем самого верхнего уровня. В HTML-коде существуют теги, которые никак не проявляются в окне браузера какими бы то ни было визуальными эффектами (например, *<HEAD>*, *<META>*, *<SCRIPT>*). Вместе с тем существует множество тегов, которым соответствуют определенные видимые элементы документа (веб-страницы).

Например, тегу *<IMG>* соответствует изображение, тегу *<BUTTON>* – кнопка, тегу *<INPUT>* – поле ввода данных, переключатель или кнопка, в зависимости от значения атрибута TYPE. Приведенный ниже HTML-код формирует простую веб-страницу, содержащую заголовок первого уровня (*<H1>*), изображение (*<IMG>*), ссылку (*<A>*) и форму (*<FORM>*), которая содержит поле ввода данных (*<INPUT>*) и кнопку (*<BUTTON>*).

```
<HTML>
<H1>Моя веб-страница</H1>
<IMG SRC="K:\Documents and Settings\LeoN\Mou документы\Mou рисунки\350.jpg">
<A HREF = "www. admiral.ru/dunaev">Сайт автора</A>
<FORM>
<INPUT TYPE="text" VALUE="">
<p>
<BUTTON>Нажми здесь</BUTTON>
</FORM>
</HTML>
```

На рис. 3.1 приведен этот HTML-документ в окне браузера.

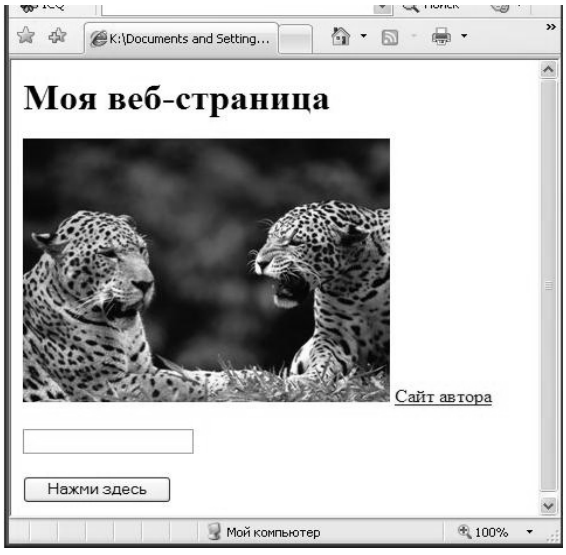

Рис. 3.1. HTML-документ в окне браузера

С помощью специальных тегов можно форматировать текстовые данные: управлять шрифтом, задавать абзацы и способ выравнивания, в том числе взаимное расположение текста и изображений.

Если не использовать специальных средств позиционирования, то элементы в документе будут располагаться в порядке их появления в HTML-коде соответствующих им тегов. При этом если не использовать теги перевода строки (<*BR*>) или абзаца (<*P*>), то элементы будут располагаться рядом друг с другом по горизонтали или по вертикали, в зависимости от размеров окна браузера и их собственных размеров. Теги <*BR*> и <*P*> являются довольно слабыми средствами форматирования. Один из часто применяемых способов разметки документа основан на использовании группы тегов, определяющих таблицу (<*TABLE*>, <*TR*>, <*TD*> и др.). Таблицы можно создавать не обязательно с видимыми границами (рамками). С помощью таблицы можно просто разбить все поле документа на прямоугольные ячейки одинаковых или различных размеров и размещать в них нужные элементы (тексты, изображения, кнопки, ссылки и т. п.). Во многих случаях этого способа разметки вполне достаточно.

§ 3.2. Динамический HTML

Выше мы бегло рассмотрели то, что называют простым, или классическим, HTML. Вскоре после его появления выяснилось, что он не вполне отвечает потребностям разработчиков и пользователей веб-сайтов. Во-первых, нужны были более мощные и гибкие средства форматирования. Во-вторых, требовалась возможность вставки в HTML-документ объектов сторонних производителей. В-третьих, требовались средства для управления содержимым HTML-документа, загруженного в браузер, а также для обработки действий пользователя (манипуляции мышью, нажатия на клавиши). Иначе говоря, было необходимо, чтобы HTML-документ являлся динамическим и интерактивным (мог взаимодействовать с пользователем).

В ответ на потребность в мощных средствах форматирования появились так называемые CSS (Cascading Style Sheets – каскадные таблицы стилей). Теперь параметры тегов стало возможным задавать не только с помощью атрибутов, но и в строке, являющейся значением специального атрибута STYLE. С помощью этого атрибута можно определить индивидуальные параметры форматирования и позиционирования практически для всех тегов. Кроме атрибута STYLE был введен в обращение контейнерный

тег <STYLE>. В теге <STYLE> можно задать индивидуальные стили для ряда тегов, а также создать произвольные стили и закрепить за ними имена. Затем для придания какому-нибудь тегу свойств поименованного стиля можно обратиться к нему с помощью атрибута CLASS = имя_стиля. С помощью стилей можно определить параметры форматирования текстов, задать фильтры (визуальные эффекты) для текстовых и графических элементов, а также задать три координаты позиционирования элементов. Позиционирование – главная функциональная возможность CSS. Координаты top и left, задаваемые с помощью стиля, определяют положение элемента на плоскости документа, а координата z-index указывает еще и слой документа. Так, задавая значения top и left для различных элементов, можно добиться того, что одни элементы будут накрывать другие. Чтобы указать, какой из них должен быть выше или ниже другого (независимо от порядка следования в HTML-коде), служит параметр z-index.

Объекты сторонних производителей, созданные с помощью различных языков программирования, вставляются в HTML-документ посредством контейнерных тегов <OBJECT> и <EMBED>. С их помощью можно встроить в HTML-документ звук, видео, Flash-анимацию, векторную графику, таблицы базы данных и многое другое.

Для управления элементами HTML-документов и даже самим браузером, генерации новых документов, организации диалогового взаимодействия с пользователем, выполнения каких-то расчетов и обработки данных в HTML была предусмотрена интеграция со специальными языками программирования. Программы, написанные на этих языках, называют сценариями (scripts). Стандартным языком сценариев является JavaScript. Его должны уметь интерпретировать все веб-браузеры. Браузер Microsoft Internet Explorer помимо JavaScript воспринимает еще один язык – VisualBasicScript, чего нельзя сказать о Netscape Navigator.

Каким образом JavaScript взаимодействует с HTML-документом и браузером? Дело в том, что браузер и загруженный в него HTML-документ представлены внутри браузера посредством иерархического множества объектов – так называемой объектной модели. Для сценария на JavaScript объекты браузера и HTML-документа образуют доступную среду. Чтобы использовать объекты браузера и документа, загруженного в него, необходимо знать их названия, свойства и методы.

§ 3.3. Работа со сценариями. Расположение сценариев

Программы, работающие с объектами HTML-документа, называют сценариями. Можно написать программу на языке JavaScript. Она может иметь самостоятельную ценность, используя собственные ресурсы JavaScript, но, кроме того, она способна взаимодействовать с объектами среды, окружающей интерпретатор языка. Интерпретатор JavaScript, встроенный в веб-браузер, предоставляет пользователю возможность использовать средства языка для доступа к ресурсам браузера и всего того, что в нем находится в данный момент. А именно к свойствам браузера и документа, загруженного в него. Конечно, вам не терпится узнать, как добраться до этих ресурсов, чтобы воздействовать на них. Это делается с помощью сценариев, написанных на языке JavaScript. Есть несколько вариантов размещения программ на JavaScript, выполняющих роль сценариев для HTML-документов. Ниже мы их и рассмотрим.

Сценарии можно писать непосредственно в HTML-документе, а также в отдельных текстовых файлах, которые вызываются из HTML-документа. Проще всего размещать сценарий непосредственно в HTML-документе, чаще всего так и поступают. Сразу заметим, что размещение сценария в отдельном файле совсем не сложно, но не всегда оправданно из экономических соображений. Необходимо накопить достаточное количество программных заготовок, чтобы решиться использовать их в последующих проектах в качестве библиотек. Размещение сценария в HTML-документе дает возможность использовать их для оперативной модификации HTML-документа, а следовательно и веб-страницы.

Стандартным является размещение сценария в контейнерном теге <*SCRIPT*>, то есть между тегами <*SCRIPT*> и </*SCRIPT*>. Встречая тег <*SCRIPT*>, браузер «понимает», что за ним начинается код сценария. Заключительный тег </*SCRIPT*> указывает браузеру, что код сценария закончился. Все, что находится вне этих тегов, браузер воспринимает как HTML-код. Контейнер <*SCRIPT*> может располагаться в любом месте HTML-документа и даже не один раз. От его расположения иногда может зависеть функционирование всего HTML-документа, но об этом мы скажем ниже. Контейнерный тег <*SCRIPT*>, объемлющий код сценария, может содержать следующие атрибуты.

LANGUAGE – язык сценария; возможные значения:

- *"JavaScript"*, *"JScript"*;
- *"VBScript"*, *"VBS"*.

Если атрибут LANGUAGE не указан, то в Internet Explorer подразумевается JScript.

SRC – указывает файл (имя или URL-адрес), содержащий код сценария. Этот атрибут используется в том случае, если сценарий расположен не в HTML-документе, а в отдельном файле.

Примеры

```
<SCRIPT LANGUAGE="JavaScript">
                    // код сценария
    </SCRIPT>
    <SCRIPT LANGUAGE = "JScript" SRC = "myscripts.JS">
</SCRIPT>
```

Редакция языка JavaScript для Internet Explorer называется JScript. Вместе с тем в Internet Explorer можно использовать и *LANGUAGE = "JavaScript"*. В браузере Netscape Navigator значимым является только *LANGUAGE = "JavaScript"*, ссылка *LANGUAGE = = "JScript"* будет им проигнорирована. Поэтому при разработке сценариев, рассчитанных для различных браузеров, рекомендуется использовать ссылку *LANGUAGE = "JavaScript"*. В примерах, приводимых в настоящей книге, мы не будем указывать язык, на котором написан сценарий (из соображений экономии места).

Если сценарий располагается в отдельном файле, то в нем, разумеется, теги *<SCRIPT>* и *</SCRIPT>* не пишутся. Как уже отмечалось, файлы со сценариями на JavaScript являются обычными текстовыми файлами. В принципе, они могут иметь любое расширение имени, но, как правило, используется расширение .js. В отдельных файлах обычно размещают библиотеки функций (определения функций), а также сценарии, использующиеся в нескольких HTML-документах одного или нескольких сайтов. Сценарий, загруженный из внешнего файла, можно представить себе просто как его вставку в HTML-документ.

Вызовы функций и их определения могут располагаться в произвольном порядке, только если они расположены в одном и том же контейнере *<SCRIPT>*. Если вы располагаете их в разных контейнерах *<SCRIPT>*, то необходимо, чтобы определение функции располагалось выше ее вызова. Аналогично, если определения

функций находятся в отдельном файле, то его необходимо загрузить в HTML-документ раньше вызовов этих функций.

При попытке загрузить и выполнить неправильный вариант HTML-кода появится диалоговое окно с сообщением об ошибке «Ошибка: Предполагается наличие объекта». Браузер интерпретирует HTML-теги последовательно. Так, встретив выражение вызова функции *myfunc()* в одном контейнере <*SCRIPT*>, браузер еще не имеет в памяти определения этого объекта (функции), расположенного в другом контейнере <*SCRIPT*>. Если же определение функции и ее вызов находятся в одном и том же контейнере <*SCRIPT*>, то его содержимое сначала загружается в память, а затем анализируется. Если интерпретатор встречает вызов функции, то он ищет ее определение в памяти и при удачном результате выполняет код этой функции.

Если необходимо, чтобы сценарий загрузился в браузер прежде, чем загрузятся элементы HTML-документа, то его следует расположить в верхней части HTML-кода. В этом случае сценарий обычно располагают в контейнере <*HEAD*> (в заголовке документа). Это самое лучшее место для расположения функций. Если требуется, чтобы сценарий загрузился после загрузки всех элементов HTML-документа, то возможны следующие два варианта. Во-первых, можно расположить сценарий в конце HTML-документа. Во-вторых, можно использовать атрибут-событие onload в контейнерном теге <*BODY*>, который задает основную часть HTML-документа. В последнем случае значением атрибута onload обычно является строка, содержащая имя функции. Определение этой функции, как правило, содержится в контейнере <*SCRIPT*>, размещенном в контейнере заголовка HTML-документа <*HEAD*>. Обратите внимание, что при использовании атрибута-события onload в теге <*BODY*> обработчик этого события выполняется по завершении загрузки всех элементов, определенных в контейнере <*BODY*>, а не в процессе их загрузки. Вот пример:

```
<HTML>
<HEAD>
<SCRIPT>
function myfunc () {
……..}
</SCRIPT>
```

```
</HEAD>
<BODY onload = "myfunc()">
……..
теги
</BODY>
……..
</HTML>
```

§ 3.4. Обработка событий

Одним из главных (но далеко не единственным) назначений сценариев в HTML-документе является обработка событий, таких как щелчок кнопкой мыши на элементе документа, помещение указателя мыши на элемент, перемещение указателя с элемента, нажатие клавиши и т. п. Большинство тегов HTML имеют специальные атрибуты, определяющие события, на которые могут отреагировать соответствующие элементы. Список всех допустимых событий довольно обширен и рассчитан практически на все случаи жизни. События называются довольно просто, особенно если вы знаете английский язык. Например, щелчок левой кнопкой мыши – onclick; изменение в поле ввода данных – onchange; событие onmouseover происходит, когда указатель мыши помещается на элемент HTML-документа. Список событий мы рассмотрим позже. Значением таких атрибутов-событий в тегах HTML является строка, содержащая сценарий, выполняющий роль обработчика события. Например, следующий HTML-код определяет заголовок второго уровня, который реагирует на щелчок кнопкой мыши тем, что выполняет некоторую функцию *myfunc()*:

```
<H2 onclick = "myfunc()">Щелкни здесь</H2>
```

Для одного и того же элемента можно определить несколько событий, на которые он будет реагировать. Другими словами, для одного и того же тега можно указать несколько атрибутов-событий. Имена этих атрибутов, как и других, можно писать в любом регистре. Порядок следования атрибутов не имеет значения.

Браузер, встречая в HTML-документе тег с определенным ID, создает в объектной модели этого документа объект с таким же именем. Для этого объекта имеется метод обработки события. Название метода совпадает с названием события, но синтаксис

использования метода требует, чтобы его название было написано в нижнем регистре. В связи с этим лучше в любом случае писать название события в нижнем регистре.

Обратите внимание на следующие два обстоятельства, важные только для рассмотренного выше способа оформления обработчиков событий.

- Элемент HTML-документа должен быть загружен раньше, чем функция-обработчик события.
- Составное имя функции-обработчика события содержит название события в нижнем регистре.

Если требуется, чтобы сценарий обработал событие независимо от того, с каким элементом оно связано, то можно воспользоваться таким составным именем функции-обработчика:

```
function document.событие(){
……….
}
```

Пример

```
function document.onclick(){
……..
}
```

Приведенных выше способов обработки событий (привязки обработчиков к элементам) вполне достаточно для практических нужд разработки веб-сайтов и других приложений. Однако следует упомянуть еще об одном способе, который применим для IE версии 4.0 и старше. Он основан на использовании атрибутов FOR и EVENT в теге <*SCRIPT*>.

Атрибуту FOR присваивается ссылка на объект, который должен реагировать на событие, а атрибуту EVENT присваивается название события. В качестве ссылки на объект обычно используется значение идентификатора объекта, то есть значение атрибута ID.

Пример

```
<HTML>
<SCRIPT FOR = "MYBUTTON" EVENT = "onclick" >
alert("Вот щелчок")
</SCRIPT>
<BUTTON ID = "MYBUTTON">Press</BUTTON>
</HTML>
```

Результат выполнения кода представлен на рис. 3.2.

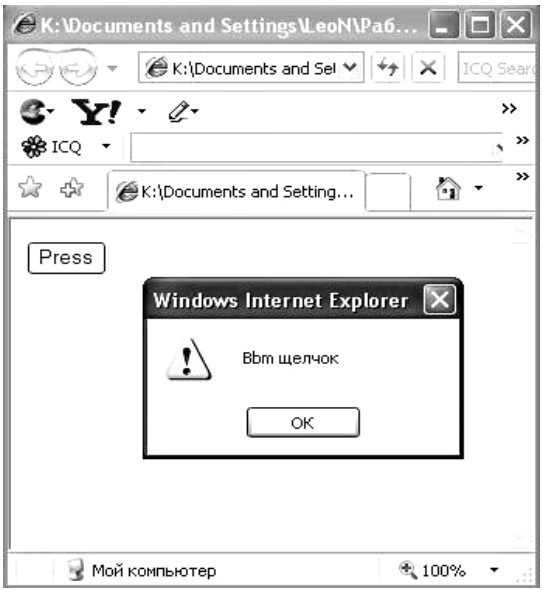

Рис. 3.2. Пример выполнения кода

Выражения в контейнере <*SCRIPT*> будут выполняться только при наступлении события, указанного в *EVENT*, которое связано с объектом, указанным в *FOR*. В приведенном выше примере при щелчке на кнопке *(<BUTTON . . .>*) будет выведено диалоговое окно с сообщением.

Рассмотренный выше способ позволяет не использовать атрибуты-события и указания обработчиков событий в тегах элементов. Однако при этом код сценария оказывается привязанным только к одному элементу. Чтобы использовать этот же код для других элементов, придется повторить его в секциях сценария (контейнерах <*SCRIPT*>), отдельно для каждого элемента. В пункте 3.6.3 мы изучим еще один способ назначения объектам обработчиков событий – с помощью сценариев.

§ 3.5. Объекты, управляемые сценариями

Мы рассмотрели, где располагаются сценарии и каким образом они могут быть привязаны к событиям. Если событие происходит, то выполняется соответствующий ему сценарий-обработчик. Кроме

того, сценарий можно запустить и вне всякой связи с каким бы то ни было событием. В любом случае сценарий должен что-то с чем-то делать. Предметом деятельности сценария являются объекты окна браузера и HTML-документа, загруженного в него. Параметры элементов документа, заданные с помощью атрибутов соответствующих тегов, можно изменить. Более того, можно заменить одни теги другими и даже заменить весь загруженный HTML-документ на другой. Делается это сценариями, но не напрямую с тегами и значениями атрибутов (то есть с HTML-кодами), а с представляющими их объектами.

HTML-документ отображается в окне браузера. Окну браузера соответствует объект window, а HTML-документу, загруженному в окно, соответствует объект document. Эти объекты содержат в своем составе другие объекты. В частности, объект document входит в состав объекта window. Элементам HTML-документа соответствуют объекты, которые входят в состав объекта document. Все множество объектов имеет иерархическую структуру, называемую объектной моделью. Более подробно мы рассмотрим ее в следующей главе, а сейчас остановимся на общих вопросах, связанных с объектами.

В предыдущих главах мы уже встречались со встроенными объектами JavaScript и объектами, создаваемыми пользователем. Напомним, что объект представляет собой своего рода контейнер для хранения информации. Он характеризуется свойствами, методами, а также событиями, на которые может реагировать. Доступ к свойствам и методам объекта осуществляется с помощью выражений вида:

объект.свойство
объект.метод()

Объекты могут находиться в отношении вложенности (подчиненности). Объект, содержащий в себе другой объект, называют родительским. Объект, который содержится в каком-нибудь объекте, называют дочерним по отношению к нему. Таким образом, устанавливается иерархия. Чтобы указать конкретный объект, требуется перечислить все содержащие его объекты начиная с объекта самого верхнего иерархического уровня, подобно тому как указывается полный путь к файлу на диске:

объект1 .объект2. . . . объектN

Если объект входит в состав другого объекта (является подобъектом другого), то для доступа к его свойствам и методам используют следующий синтаксис:

объект1.объект2 . . . . объектN. свойство
объект1.объект2. . . . объектN.метод()

Нередко подобъект некоторого объекта называют его свойством (так сказать, сложным свойством). В этом случае можно говорить, что свойства объектов бывают трех типов:

- просто свойства (простые свойства);
- методы (свойства-функции);
- объекты (сложные свойства, имеющие свои свойства).

В объектной модели документа объекты сгруппированы в так называемые коллекции. Коллекцию можно рассматривать как промежуточный объект, который содержит объекты собственно документа. С другой точки зрения, коллекция является массивом объектов, отсортированных в порядке упоминания соответствующих им элементов в HTML-документе. Индексация объектов в коллекции начинается с нуля. Синтаксис обращения к элементам коллекции такой же, как к элементам массива. Коллекция даже имеет свойство length – количество всех ее элементов. Так, например, коллекция всех объектов графических изображений в документе называется images, коллекция всех форм – forms, коллекция всех ссылок – links. Это примеры так называемых частных или тематических коллекций. Кроме них имеется коллекция всех объектов документа, которая называется all. Один и тот же объект может входить в частную коллекцию (например, images), но обязательно входит в коллекцию all. При этом индексы этого объекта в частной коллекции и коллекции all могут быть различными.

Рассмотрим способы обращения к свойствам объектов документа. Общее правило заключается в том, что в ссылке должно быть упомянуто название коллекции. Исключением из этого правила является объект формы. Далее, если речь идет о документе, загруженном в текущее окно, то объект window можно не упоминать, а сразу начинать с ключевого слова document. Возможны несколько способов обращения к объектам документа:

- *document.коллекция ID_объекта;*
- *document.коллекция["ID_объекта"];*
- *document.коллекция[индекс_объекта].*

Здесь ID_бъекта – значение атрибута ID в теге, который определяет соответствующий элемент в HTML-документе. Величина индекс_объекта – целое число, указывающее порядковый номер объекта в коллекции. Первый объект в коллекции имеет индекс 0. Если при создании HTML-документа вы не использовали атрибут ID для некоторых элементов, то для обращения к их объектам остается только взять индексы. Заметим, что некоторые старые теги (например, *<FORM>*, *<INPUT>*) имеют атрибут *NAME*. Значение этого атрибута можно использовать при обращении к объекту наравне со значением атрибута ID. К объекту формы, кроме описанных выше способов, можно обращаться по значению атрибута *NAME* (имя формы), если он указан в теге *<FORM>*, но не по значению атрибута ID:

document.имя_формы

Как известно, тег формы *<FORM>* является контейнерным и может содержать в себе теги других элементов, такие как *<INPUT>*, *<BUTTON>* и др. Обращение к элементам формы возможно через коллекцию all. Однако имеется и специфический для них способ:

document.имя_формы.elements[индекс_элемента]
document.имя_формы.elements[id_элемента]
document.имя_формы.id_элемента

Здесь индекс_элемента – порядковый номер элемента формы, а не порядковый номер в коллекции всех элементов; первый элемент имеет индекс 0. Заметим, что для Internet Explorer вместо квадратных скобок допустимо использование и круглых скобок. Приведенный ниже HTML-код формирует простую веб-страницу, содержащую заголовок первого уровня, изображение, ссылку и форму с двумя элементами, полем ввода данных и кнопкой.

```
<HTML>
<H1>Моя веб-страница</H1>
<IMG SRC = "K:\Documents and Settings\LeoN\Мои документы\Мои рисунки\ 350.jpg" >
<A HREF = "www.admiral.ru/~dunaev">Сайт автора</A>
<FORM>
<INPUT TYPE = " text" VALUE = " " >
<p>
```

```
<BUTTON onclick = "my()">Нажми здесь</BUTTON>
</FORM>
</HTML>
```

Результат выполнения кода представлен на рис. 3.3.

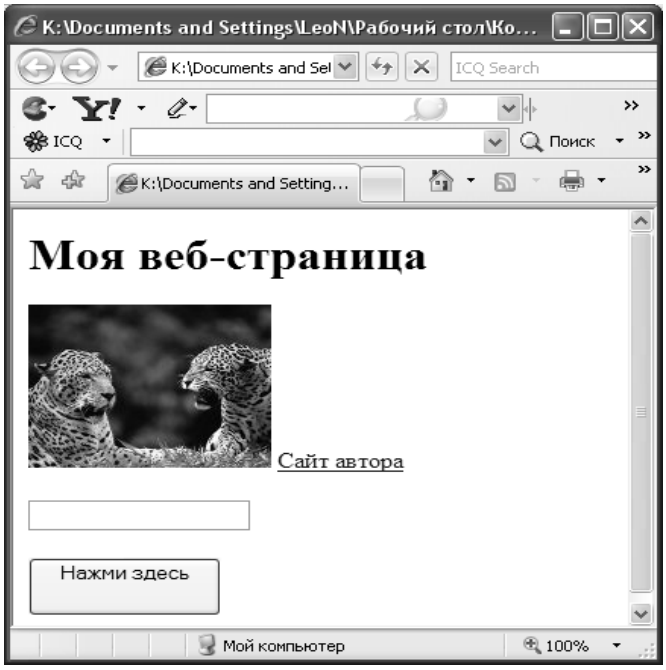

Рис. 3.3. Пример выполнения кода

Поскольку ни для одного из элементов документа не указан атрибут ID (или NAME), обращение к объектам возможно посредством индексов. Так, например, для обращения к объекту графического объекта (он у нас единственный) можно использовать следующее выражение

```
document.images(0)   // первый элемент коллекции images
```

Кнопка является вторым элементом формы. Чтобы обратиться к ней как к объекту, необходимо указать сначала форму (единственная форма в документе есть объект forms(0)), а затем объект этой формы (elements(индекс_элемента)). Другими словами, достаточно записать выражение:

```
Document.forms(0).elements(1)          // второй элемент формы
```

Теперь обратимся к объекту ссылки в нашем документе. Все ссылки хранятся в коллекции links. В нашем случае коллекция

links содержит лишь одну ссылку. Следовательно, наша ссылка – элемент links(0) массива links. В итоге для обращения к ссылке в HTML-документе достаточно написать следующее выражение:

document.links(0)

Обобщая вышеизложенное, отметим, что универсальный способ обращения к объектам документа – обращение посредством коллекции all. Сценарий основан на использовании свойства *tagName*, которое возвращает название тега, с помощью которого был создан соответствующий объект. Код сценария приведен в листинге 3.1.

Листинг 3.1. Тестируемый HTML-код
вместе с тестирующим сценарием

```
<HTML>
<H1> Моя веб-страница</H1>
<IMG SRC = "K:\Documents and Settings\LeoN\Мои документы\Мои рисунки\Зверушки\350.jpg">
<A HREF = "www. admiral.ru/~dunaev">Сайт автора</A>
<p>
<FORM>
<INPUT TYPE = "text" VALUE = "">
<p>
<BUTTON onclick = "my()">Нажми здесь</BUTTON>
</FORM>
<SCRIPT>
msg = ""
for(i = 0; i < document.all . length; i++){
msg += i + " " + document . all [i] . tagName + "\n"
}
alert ( msg)
</SCRIPT>
</HTML>
```

В результате выполнения этого HTML-кода окно браузера будет выглядеть, как показано на рис. 3.4. В диалоговом окне, выведенном с помощью *alert()*, перечислены теги HTML-документа. Их порядковые номера соответствуют индексам в коллекции all. Например, объекту *document.all(5)* соответствует элемент HTML-документа, созданный тегом *<IMG>*. Обратите внимание, что хотя теги

<*HEAD*>, <*TITLE*> и <*BODY*> в нашем документе явно не упоминаются, соответствующие им объекты присутствуют в объектной модели этого документа и, следовательно, в коллекции all.

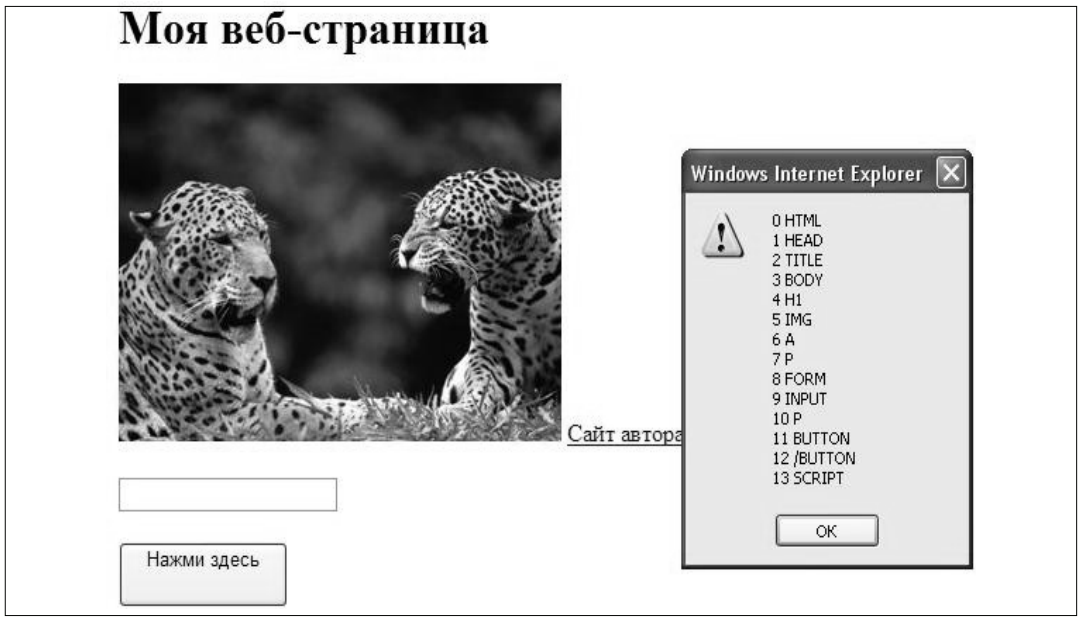

Рис. 3.4. В диалоговом окне перечислены теги HTML-документа и их индексы в коллекции all

Теперь рассмотрим тот же пример, но добавим к основным тегам атрибуты ID, с помощью которых можно персонифицировать соответствующие объекты. Как и раньше, добавим к HTML-документу сценарий, выводящий окно с информацией о тегах. При этом, кроме порядкового номера и названия тега, будем выводить и значение атрибута ID (рис. 3.5). Чтобы получить значение атрибута ID, необходимо воспользоваться свойством id соответствующего объекта. HTML-код приведен в листинге 3.2.

Листинг 3.2. Тестируемый код с добавлением атрибутов ID

```
<HTML>
<H1>Моя веб-страница</H1>
<IMG ID = "myimage" SRC = "K:\Documents and Settings\LeoN\Mou документы\Мои рисунки\350.jpg" >
<A ID = "myref" HREF = "www.admiral.ru/~dunaev">Сайт автора</A>
<FORM ID = "myform">
```

```
<INPUT ID = "myinput" TYPE = "text" VALUE ="qewe">
<p>
<BUTTON ID = "mybutton" onclick = "my() ">Нажми здесь
</BUTTON ID>
</FORM>
<SCRIPT>
msg = " "
for(i =0; i<document.all.length; i + t) {
msg+ = i + " " + document.all[i].tagName + " id = " + docu-
ment.all[i].id + "\n"
}
alert(msg)
</SCRIPT>
</HTML>
```

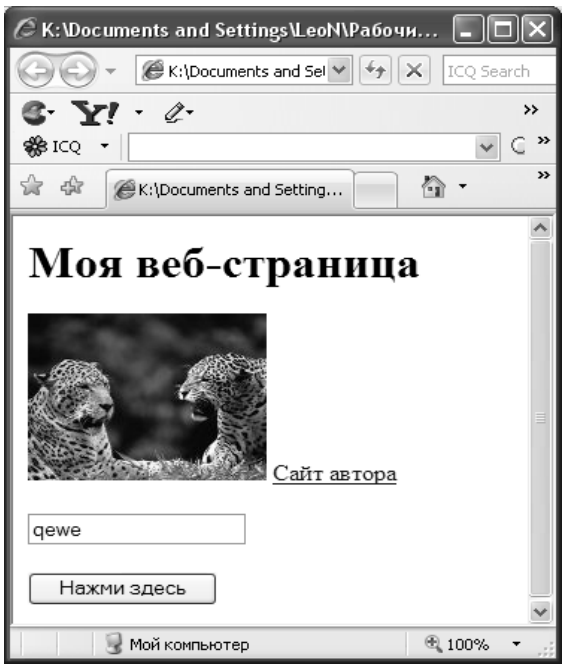

Рис. 3.5. Теги HTML-документа, их индексы
в коллекции all, а также значения атрибутов ID

Итак, универсальный способ обращения к объекту документа базируется на использовании коллекции all. Однако кроме него в IE5+ и NN6 существует еще один способ, основанный на методе *getElementByldQ*:

document.getElementBy Id(значение_ID)

Обратите внимание, в каком регистре пишутся отдельные буквы этого метода. Метод *getElementById(значение_ID)* позволяет обратиться к любому элементу по значению его идентификатора – значению атрибута ID. Если несколько элементов документа имеют одинаковый ID, метод возвращает первый элемент с указанным значением ID. Отсюда видно, насколько важно присваивать различным элементам уникальные значения ID. Выше мы рассмотрели различные способы обращения к объектам. Ссылки на объекты часто используются, главным образом, как промежуточный этап для доступа к конечным свойствам и методам. В конце концов, именно они и интересуют нас при создании сценариев. Заметим попутно, что ссылки на объекты можно сохранять в переменных, которые потом можно использовать для обращения к свойствам и методам.

В приведенном выше примере элемент изображения, созданный с помощью тега *<IMG ID = "myimage" SRC = "pict.jpg">*, имеет атрибуты *ID* и *SRC*. Этим атрибутам соответствуют свойства id и src объекта этого изображения. Ниже приведены примеры различных вариантов обращения к данным свойствам:

```
document.images(0).id      //"myimage"
document. images ("myimage") . id
document.all("myimage").id
document.all.myimage.id
var x = document.all.myimage // объект
x.id                       // "myimage"
document.images(0).src                  /* строка URL, например,
file:///C:/Мои документы/pict.jpg */
document.images("my image").src
document.all("myimage").src
document.all.myimage.src
var x = document.all.myimage       // объект
x.src                              /* строка URL, например, file:
///C:/Мои документы/pict .jpg */
```

Объекты документа имеют много интересных и полезных свойств. Одни из них доступны только для чтения и могут использоваться в сценариях в качестве информации для выполнения каких-то действий. Другие свойства можно изменять, то есть присваивать им иные значения (например, для изменения рисунка достаточно изменить значение свойства src). В следующем параграфе мы рассмотрим, для чего служат конкретные свойства и методы.

§ 3.6. Понятие события

Каждое действие пользователя (нажатие на клавишу, щелчок кнопкой мыши и т. п.) формирует некоторое событие, то есть сообщение о произошедшем. Операционная система (например, Windows) анализирует это сообщение, чтобы узнать, откуда оно взялось и что с ним делать дальше. Если, например, пользователь нажал на кнопку мыши в момент, когда ее указатель находился над окном браузера, то Windows пошлет браузеру сообщение о том, какая кнопка мыши была нажата, какие при этом клавиши клавиатуры удерживаются, а также координаты указателя мыши. Если пользователь щелкнул где-то на панели инструментов, браузер отработает это сообщение сам. Если же в момент щелчка указатель находился на «территории» HTML-документа, то браузер пропустит сообщение о событии через свою объектную модель. В HTML-коде документа может находиться сценарий для обработки этого события. Инструкции этого сценария направляются к браузеру опять же через объектную модель.

3.6.1. Свойства события

Сообщение о событии формируется в виде объекта, то есть контейнера для хранения информации. Как только объект события создан, браузер присваивает значения его свойствам. Например, объект, соответствующий щелчку мышью, содержит координаты указателя мыши, а также сведения о том, какая кнопка была нажата. Кроме того, объект события в одном из своих свойств содержит ссылку на элемент, с которым связано данное событие (например, на кнопку, изображение, поле ввода и т. п.).

Объект события хранится в памяти столько времени, сколько необходимо сценарию для его обработки. Пока выполняется обработчик события, объект события вместе со своими свойствами доступен сценарию, находящемуся в памяти браузера. Как только обработчик события завершит работу, объект события становится пустым (возвращается в исходное состояние).

В любой момент времени существует не более одного объекта события. Все инициированные события заносятся операционной системой в буфер и выполняются последовательно в том порядке, в каком они туда попали.

В объектной модели имеется объект *event*, являющийся подобъектом объекта окна window. Он содержит информацию о том,

какое событие произошло, какой элемент должен на него реагировать, и ряд других характеристик. Объект event можно использовать в сценариях для документов с большим количеством интерактивных элементов, для выяснения причин неправильного отображения веб-страниц. Далее мы рассмотрим некоторые наиболее часто используемые свойства объекта event.

Свойство *srcElement* возвращает ссылку на объект элемента HTML-документа, который инициировал событие. При получении такой ссылки можно узнать или изменить значения свойств этого объекта и применить к нему любой из его методов.

```
<HTML>
<BODY onclick = "changetext()">
<button>Первая кнопка</button>
<button>Вторая кнопка</button>
</BODY>
<SCRIPT>
function changetext(){
x = window.event.srcElement          // ссылка на объект
x.innerText = "Уже щелкнули"         // замена текста
}
</SCRIPT>
</HTML >
```

Результат выполнения кода представлен на рис. 3.6.

Чтобы на щелчок реагировали только кнопки, приведенный выше код необходимо несколько модифицировать:

```
<HTML>
<BODY>
<button onclick = "changetext()">Первая кнопка</button>
<button onclick = "changetext()">Вторая кнопка</button>
</BODY>
<SCRIPT>
function changetext(){
x = window.event.srcElement                  // ссылка на объект
x.innerText = "Уже щелкнули"                 // замена текста
}
</SCRIPT>
</HTML>
```

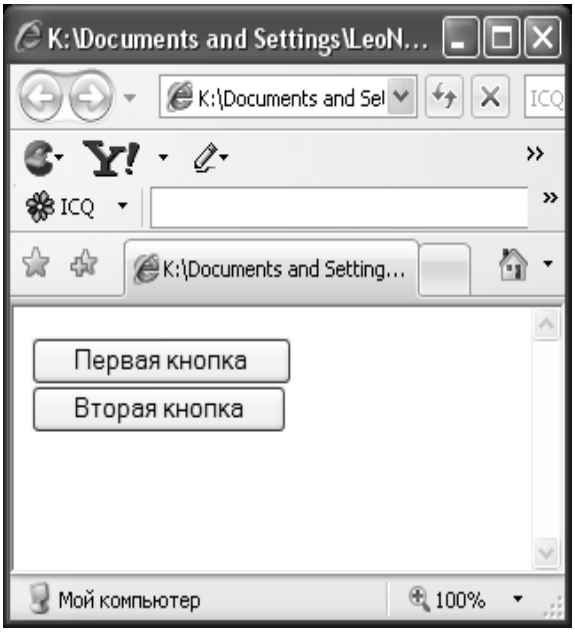

Рис. 3.6. Пример выполнения кода

Объект event имеет несколько свойств, содержащих координаты в пикселах указателя мыши в момент возникновения события, связанного с мышью. Многообразие типов этих координат обусловлено различиями в начале их отсчета. Рассмотрим эти свойства по порядку.

• screen*X*, screen*Y* – координаты указателя мыши относительно левого верхнего угла экрана. Например, если экран монитора находится в режиме 800×600, то координаты правого нижнего угла экрана будут равны 800 и 600 пикселов соответственно.

• client*X*, client*Y* – координаты указателя мыши относительно области клиента (рабочей области браузера), в которой находится HTML-документ, без учета рамок, полос прокрутки, меню и т. п.

• offset*X*, offset*Y* – координаты указателя мыши относительно верхнего левого угла элемента, инициировавшего событие.

• *x*, *y* – координаты указателя мыши относительно верхнего левого угла первого абсолютно или относительно позиционированного контейнера, в котором находится элемент, инициировавший событие. Позиционированный контейнер – это элемент, заданный каким-нибудь контейнерным тегом (например, <BODY>, <DIV>, <H1>) с атрибутом STYLE, для которого указано свойство position. Если такого контейнера нет, то свойства *x* и *y* возвращают координаты относительно главного документа, такие же, как и client*X*, client*Y*.

У объекта event имеется изменяемое свойство returnValue (возвращаемое значение). С его помощью можно отменять действия, принятые по умолчанию. Для этого достаточно присвоить ему значение false. Например, щелчок на ссылке по умолчанию означает переход по указанному адресу. Однако можно отменить это действие или запросить у пользователя подтверждение перехода.

Пример

```
<HTML>
<A ID = "myref " HREF = "www.chat.ru">Переход на chat.ru</A>
<SCRIPT>
function myref.onclick(){
ret =confirm(Вы действительно хотите перейти?")
if (Iret)
window.event.returnValue = false
}
</SCRIPT>
</HTML>
```

Результат выполнения кода представлен на рис. 3.7.

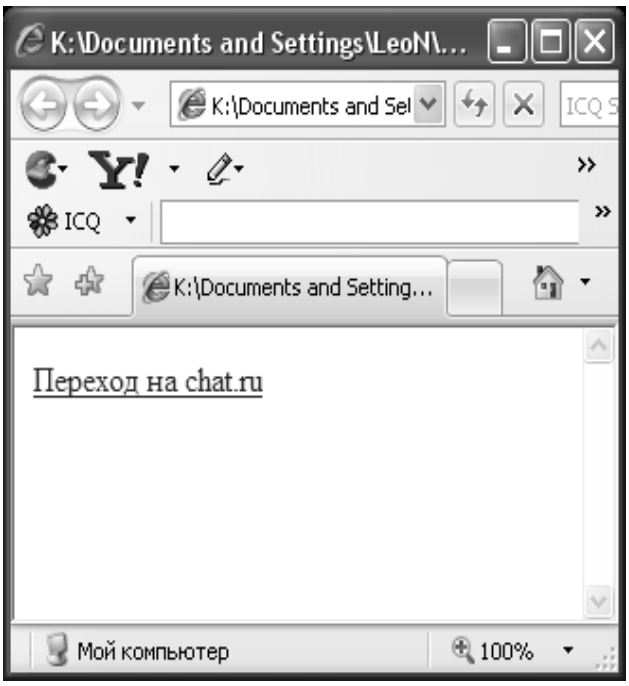

Рис. 3.7. Пример выполнения кода

Здесь метод *confirm()* выводит диалоговое окно. Если пользователь не подтвердил переход, то *confirm()* возвращает false, а свойству returnValue тоже присваивается false. В результате переход по ссылке не произойдет. Нередко требуется разместить в документе элемент, который внешне выглядит как обычная ссылка (при наведении указателя мыши его вид изменяется), но щелчок должен приводить не к переходу на другой документ, а просто к выполнению некоторого сценария. Этого можно добиться несколькими способами. Один из них основан на использовании свойства returnValue:

```
<HTML>
<A ID = "myref" HREF = "">Щелкни здесь</A>
<SCRIPT>
function myref.onclick(){
alert ("5bm щелчок на ссылке")
window.event.returnValue = false
}
</SCRIPT>
</HTML>
```

Результат выполнения кода представлен на рис. 3.8.

Рис. 3.8. Пример выполнения кода

Если бы мы не использовали оператор *window.event.returnValue = false*, то из-за неуказанного в *HREF* адреса перехода по ссылке после выполнения обработчика события открылось бы новое окно браузера с содержимым текущей папки. Другой способ отключить действие щелчка на ссылке, принятое по умолчанию, заключается в том, чтобы в значении атрибута *HREF* написать, как в следующем примере:

```
<A ID = "myref" HREF = "#" onclick = "myclick()">Щелкни здесь</A>
```

3.6.2. Прохождение событий

Как уже отмечалось выше, для элементов документа можно указать события, на которые они должны реагировать, и обработчики этих событий. Например, если требуется, чтобы элемент реагировал на щелчок кнопкой мыши выполнением некоторой функции *myfunc()*, то в тег этого элемента следует вставить запись: *onclick = "myfunc()"*. Однако, нам известно, что большинство тегов в HTML являются контейнерными и, следовательно, могут содержать в себе другие теги. При этом может оказаться так, что одно и то же событие будет обозначено в различных, но вложенных друг в друга тегах. Что произойдет при наступлении этого события? Как оно распространяется по объектам и как оно перехватывается элементами документа? Рассмотрим в качестве примера следующий HTML-документ, содержащий единственную кнопку:

```
<HTML>
<BODY onclick = "alert('Щелчок на body')">
<BUTTON onclick = "alert('Щелчок на button') ">Нажми здесь</BUTTON>
</BODY>
</HTML>
```

Особенность этого документа в том, что одно и то же событие (с различными обработчиками) привязано к различным тегам, один из которых содержит в себе другой (*<BUTTON>* содержится в *<BODY>*). Щелчок на кнопке приведет к выполнению сначала обработчика щелчка для кнопки, а затем обработчика для *<BODY>*. Если щелкнуть где-нибудь в рабочей области окна браузера вне кнопки, то сработает только обработчик для *<BODY>*. Ниже мы рассмотрим, как можно избавиться от этого нежелательного эффекта.

Чтобы события обрабатывались на уровне текущих объектов, требуется включить для них режим захвата. Это делается с помощью специального метода *captureEvent()*. В IE4 для управления всплыванием событий используют свойство прерывания всплывания *cancelBubble*.

В современных браузерах IE5.5+ и NN6 реализована модель, объединяющая в себе и всплывание, и захват событий. Всплывание организовано как в IE4+, а захват – как в NN4. По умолчанию событие всплывает, однако можно включить его захват. Тогда событие сначала достигает целевого объекта, а затем начинает всплывать в обратном направлении. В IE4+ управлять прохождением событий можно с помощью свойства *cancelBubble* объекта события event. Когда в документе (на веб-странице) происходит какое-либо событие, объект event первым получает информацию о нем и решает, какому элементу его передать. Например, когда объект *event* получает событие onclick (щелчок кнопкой мыши), он выясняет, какой элемент находился в тот момент под указателем мыши. Если там было два элемента (один над другим), то выбирается элемент с наименьшим z-индексом, то есть нижний. Как говорилось ранее, что z-index является одним из параметров позиционирования элементов с помощью таблиц стилей. Установив элемент, связанный с событием, объект event ищет обработчик этого события и выполняет его. Например, если элемент имеет уникальное имя (ID) Myelement, то ищется функция *Myelement.onclick()*. Далее объект event выясняет, какой объект является контейнером для данного элемента. Если таковой имеется, то событие переходит внутрь этого контейнера. Если, например, контейнером является объект, созданный тегом *<DIV>* с именем Mydiv, то будет предпринята попытка выполнить функцию *Mydiv.onclick()*. Этот процесс продолжается, пока имеются доступные контейнеры. Например, последним доступным контейнером может оказаться документ (объект document), и тогда объект *event* ищет функцию *document.onclick()*.

Чтобы определить главного виновника события (объект, инициировавший событие), используется свойство *srcElement*. Если потребуется одинаковая реакция на одно и то же событие всех элементов, за исключением некоторых, в таких случаях для элементов, на которых процесс всплывания события должен закончиться, создаются специальные функции-обработчики. В этих

функциях помимо прочего должно быть выражение присвоения свойству *cancelBubble* значения true, чтобы прервать дальнейшее прохождение события. Например, если мы хотим разорвать цепную передачу события на элементе с именем *Myelement*, необходимо создать для него функцию-обработчик события следующего вида:

```
function Myelement. Onclick(){
        // код обработки события onclick
window.event. cancelBubble = true
}
```

Делать такое в функции-обработчике для объекта document бессмысленно, поскольку выше событие все равно всплыть не может, а отменить реакцию на это событие элемента-инициатора (первого звена цепочки) невозможно.

В начале этого подраздела мы уже рассматривали пример HTML-документа с кнопкой, вложенной в контейнер <*BODY*>:

```
<HTML>
<BODY onclick = "alert('Щелчок на body')">
<BUTTON onclick = "alert ('Щелчок на button')">Нажми здесь</BUTTON>
</BODY>
</HTML>
```

Результат выполнения кода представлен на рис. 3.9.

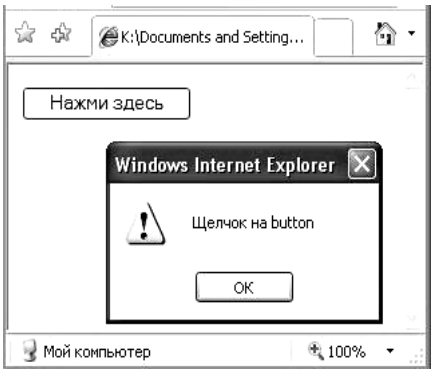

Рис. 3.9. Пример выполнения кода

Особенность этого примера в том, что если пользователь щелкнет на кнопке, сработает обработчик не только кнопки, но и тела документа (элемента, заданного тегом <*BODY*>. Если мы хотим это предотвратить, то можно переписать код следующим образом:

```
<HTML>
<BODY onclick = "alert('Щелчок на body')">
<BUTTON onclick = "alert('Щелчок на button') ;
window.event .cancelBubble = true ">Нажми здесь</BUTTON>
</BODY>
</HTML>
```

Возможен также и такой вариант кода:

```
<HTML>
<BODY onclick = "alert('Щелчок на body')">
<BUTTON ID ="Mybut">Нажми здесь</BUTTON>
</BODY>
<SCRIPT>
function Mybut.onclick() {
alert('Щелчок на button')
window.event.cancelBubble = true /* прекратить всплывание события */
I
</SCRIPT>
</HTML>
```

3.6.3. Указание обработчика события в сценарии

Об обработчиках событий достаточно подробно рассказывалось в предыдущем параграфе. Там так или иначе обработчики событий привязывались к специальным атрибутам HTML-тегов – ID или атрибутам-событиям – *FOR* и *EVENT*. Сценарий-обработчик выполнялся, если возникало событие, связанное с элементом, для которого был указан этот обработчик. Вместе с тем имеется возможность вызывать обработчик и сценарии просто как метод объекта. Он будет выполняться так же, как и при возникновении соответствующего события.

Пример

```
<HTML>
<BUTTON ID="Mybutton" onclick = " butclick() ">Нажми здесь</BUTTON>
<SCRIPT>
```

```
Document.all.Mybutton.onclick      // вызов обработчика
function butclick(){
alert('Щелчок на кнопке Mybutton')
I
</SCRIPT>
</HTML>
```

В данном случае сразу же после загрузки документа в браузер вызывается обработчик события onclick, связанного с кнопкой. Этот обработчик – функция *butclick()*. Обратите внимание, что обработчик выполнится несмотря на то, что самого щелчка не было. Заметим, что выражение вызова метода (в данном случае это *onclick()*) должно записываться только строчными буквами.

Следует иметь в виду, что рассмотренный выше способ использования обработчика как метода существенно отличается от эмуляции действия для события. Например, допустим, что имеется форма с несколькими полями для ввода данных, одно из которых имеет обработчик события onfocus (активизация поля). Чтобы поле активизировалось, необходимо щелкнуть на нем мышью или перейти к нему с помощью клавиатуры. При этом сработает обработчик события onfocus. Однако даже если поле не активно, обработчик события onfocus все равно можно вызвать, но при этом поле не активизируется:

```
document.имя_формы.имя_поля.onfocus()
```

Благодаря возможности использовать обработчик события в качестве метода объекта, можно задавать обработчик в виде функции, имеющей название, совпадающее с названием события:

```
<HTML>
<BODY onload = "onload()"
……..
</BODY>
<SCRIPT>
function onload() {
</SCRIPT>
</HTML>
```

Практические задания к главе 3

Задание 1

Создайте два фрейма, расположенные друг над другом:

```
<HTML>
<FRAMESET ROWS="30%,70%">
<FRAME SRC="документ1" NAME="frame1">
<FRAME SRC="документ2" NAME="frame2">
</FRAMESET>
</HTML>
```

Результат выполнения кода представлен на рис. 3.10.

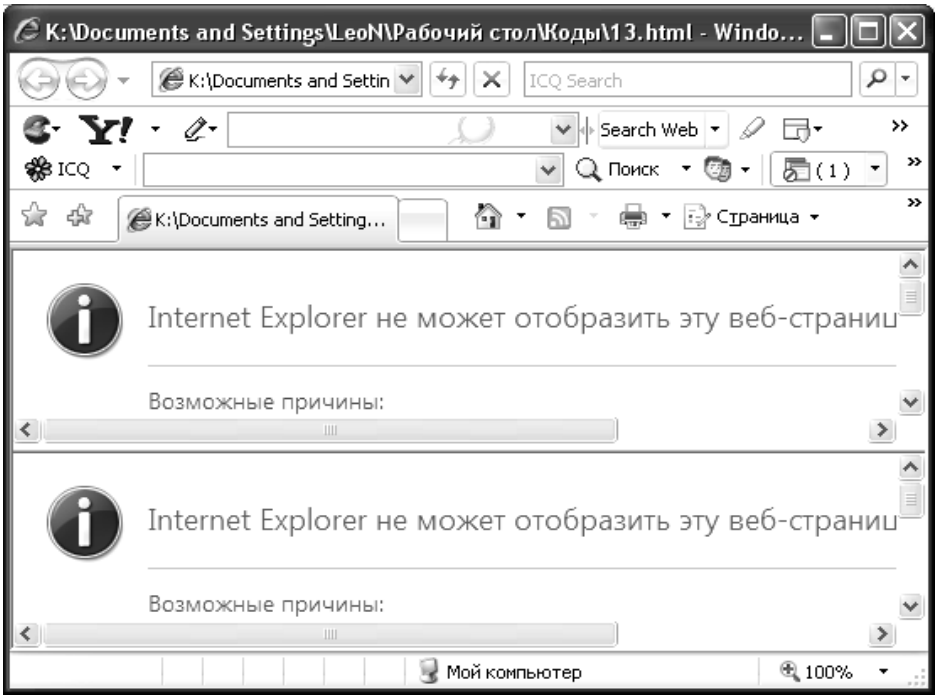

Рис. 3.10. Пример выполнения кода

Это так называемое вертикальное расположение фреймов. Если вместо атрибута *ROWS* в теге *<FRAMESET>* использовать атрибут *COLS*, то фреймы будут расположены горизонтально: второй фрейм находится справа от первого. Используя вложение тега *<FRAMESET>*, можно разбить уже имеющийся фрейм на другие два фрейма и т. д.

Задание 2

Разделить окно браузера на два фрейма: первый выполняет роль навигационной панели, а второй – окна для отображения документов. frames. htm – установочный файл.

```
<HTML>
<FRAMESET COLS = "25%,75%">
<FRAME SRC = " menu.htm " NAME = "menu">
<FRAME SRC = " start.htm " NAME = " main " >
</FRAMESET>
</HTML>
```

Результат выполнения кода представлен на рис 3.11.

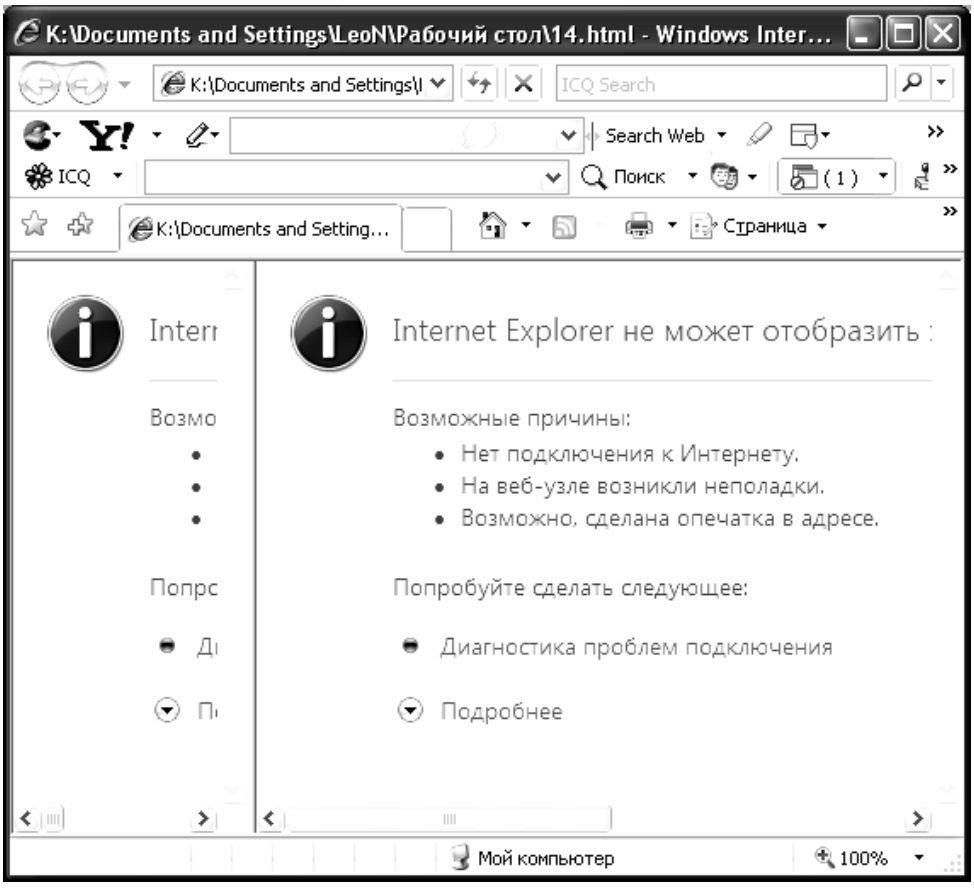

Рис. 3.11. Пример выполнения кода

Задание 3

Создать HTML-документ с двумя кнопками, щелчок на которых вызывает одну и ту же функцию pop(xcontent), открывающую

всплывающее окно и заполняющую его содержимым, которое задано параметром *xcontent*:

```
<HTML>
<H3>Vsplivaus4ie okna</H3>
<BUTTON onclick = "pop('Hello') ">Prosto tekst</BUTTON>
<BR><BR>
<BUTTON onclick = "pop('<IMG SRC = K:\Documents and Settings\LeoN\Мои документы\Мои рисунки\350.jpg>Leopardy ') ">Kartinka s tekstom</BUTTON>
<SCRIPT>
var mypopup = window. createPopup()
var popupBody = mypopup.document.body
popupBody.style.border = "solid 2px green"
popupBody.style.padding = "5px"
popupBody.style.color = "blue"
popupBody.style.background = "ffffd0"
function pop(xcontent) {
popupBody.innerHTML = "<p>"+ xcontent + "</p>"
mypopup.show(130,90,400,200,document.body)
}
</SCRIPT>
</HTML>
```

Поэкспериментировав с фреймами, вернемся к нашему старому проекту.

Задание 4

Написать функцию, которая открывает ссылку при наведении на нее курсора мышки, без щелчка.

Предлагаемый код

```
<a href="html.html" target=main onmouseover="window.open ('html.html');"><FONT SIZE=7>Сайт автора</FONT></a>
```

Примечание. Большинство браузеров без каких-либо проблем блокируют данный код. Некоторые блокируют автоматически. Протестировать можно в том же IE, разрешив заблокированное содержимое. Используется нечестными программистами для раскрутки сайтов.

Задание 5

В некоторых случаях приходится ставить ограничение по возрасту на своем сайте. Ясное дело, что мало кто честно ответит на вопрос, если он хочет добиться желаемого результата. Но формальности, как говорится, будут соблюдены.

Предлагаемый код

```
<SCRIPT LANGUAGE="javascript">
function getName() {
        var old = prompt("Пожалуйста, укажите ваш возраст:", "")
        if( old < 18) {
   if (old == ''){ old = 0;}
                alert("Простите, но вам всего " + old + " лет, а этот сайт разрешено просматривать только пользователям старше 18-ти лет ")
        window.open("http://www.woweb.ru")
        } else {
                alert("Вам " + old + " лет. Прекрасно. Смотрите и наслаждайтесь!")
        }
}
</SCRIPT>
```

Глава 4
РАБОТА С ОКНАМИ И ФРЕЙМАМИ

Как известно, HTML-документ загружается в окно браузера. Можно открыть несколько таких окон и загрузить в них различные документы, а также разбить одно окно на несколько прямоугольных областей, называемых фреймами. В каждый такой фрейм можно загрузить отдельный документ. При этом существует возможность организовать взаимодействие между фреймами – например, с помощью действий в одном фрейме управлять содержимым другого фрейма.

Загрузка в браузер HTML-документа приводит к тому, что в браузере создается иерархическая объектная модель этого документа, на самом верхнем уровне которой находится объект window. Доступ к свойствам и методам данного объекта имеет уже знакомый вам синтаксис:

window.свойство
window.метод([параметры])

У объекта *window* имеется синоним *self*, используемый при обращении к окну, содержащему текущий документ. Иначе говоря, идентификатор *self* применяется в многооконных или многофреймовых системах, когда требуется указать окно с документом, в котором находится данный сценарий. Его рекомендуется вставлять, чтобы не запутаться. При запуске сценария в ссылках на объекты только текущего документа (типичная ситуация) идентификаторы *window* и *self* можно опускать.

У объекта *window* есть ряд подобъектов. Мы уже рассматривали некоторые свойства объекта события event, который является подобъектом объекта *window*. Другой *объект*, *location*, содержит информацию, полезную для работы в сети и для создания ссылок в документах с многофреймовой структурой. Кроме этого, свойство href объекта location используется для загрузки документа в текущее окно:

window.location.href = "URL-адрес_документа"

Данный способ загрузки документов в текущее окно браузера доступен во всех версиях IE и NN. В IE можно использовать также и метод *navigate()*:

window.navigate ("URL-адрес_документа ")

§ 4.1. Создание новых окон

Главное окно браузера создается не с помощью сценариев, а автоматически, когда пользователь запускает браузер, а также при открытии документа с определенным URL-адресом или другого файла. В HTML открыть документ в новом окне можно с помощью атрибута TARGET тега ссылки *<A HREF = . . . >*. Например, *<A HREF = "http://www.rambler.ru" TARGET = "newWindow"> Rumbler</A>*.

С помощью сценария можно создать любое количество окон. Для этого применяется метод *open()*:

window.open([параметры])

Этому методу передаются следующие необязательные параметры:

- адрес документа, который нужно загрузить в создаваемое окно;
- имя окна (как имя переменной);
- строка описания свойств окна (features).

В строке свойств записываются пары свойство= значение, которые отделяются друг от друга запятыми. В табл. 4.1 приведен список свойств окна, передаваемых в строке features. Значения yes и no можно заменить числовыми эквивалентами 1 и 0 соответственно.

Таблица 4.1

Свойства окна, передаваемые в строке features

Свойство	Значения	Описание
channel mode	yes, no, 1, 0	Показывает элементы управления Channel
directories	yes, no, 1, 0	Включает кнопки каталога
fullscreen	yes, no, 1, 0	Полностью разворачивает окно
height	Число	Высота окна в пикселах
left	Число	Положение по горизонтали относительно левого края экрана в пикселах

Окончание табл. 4.1

Свойство	Значения	Описание
location	yes, no, 1, 0	Текстовое поле Address
menubar	yes, no, 1, 0	Стандартные меню браузера
resizeable	yes, no, 1, 0	Может ли пользователь изменять размер окна
scrollbars	yes, no, 1, 0	Горизонтальная и вертикальная полосы прокрутки
status	yes, no, 1, 0	Стандартная строка состояния
toolbar	yes, no, 1, 0	Включает панели инструментов браузера
top	Число	Положение по вертикали относительно верхнего края экрана в пикселах
width	Число	Ширина окна в пикселах

Примеры

window.open("mypage.htm","NewWin", "height=150, width=300")

window.open("mypage.htm")

strfeatures = "top=100,left=15,width=400, height=200, location=no,menubar=no"

window.open("www.admiral.ru/~dunaev", "Сам себе веб-дизайнер"

strfeatures)

Вместо третьего параметра (строки features) можно использовать значение true. В этом случае указанный документ загружается в уже существующее окно, вытесняя предыдущий. Например, *window.open("mypage.htm","NewWin", true)*.

Метод *window.open()* возвращает ссылку на объект окна. Эту ссылку можно сохранить в переменной, чтобы потом использовать, например, при закрытии окна. Для закрытия окна служит метод *close()*. Однако выражения *window.close()* или *self.close()* закрывают главное окно, а не дополнительное, которое вы создали методом *open()*. В этом случае как раз и необходима ссылка на созданное окно. Эту ссылку следует сохранить в глобальной переменной, чтобы иметь доступ к ней до тех пор, пока главный документ загружен в браузер, например:

```
var objwin = window. open ("mypage. htm", "Моя страница")
objwin.close()
```

Метод *window.open()* открывает новое независимое окно как экземпляр браузера. В этом случае при закрытии главного окна браузера новое окно остается открытым. Независимые окна называют еще немодальными (modalless). Однако можно создать и так называемое модальное окно. Пока открыто модальное окно, пользователь не может обратиться к другим окнам, в том числе и к главному. Так обычно работают стандартные диалоговые окна. Например, окна, создаваемые методами *alert()*, *prompt()* и *confirm()*, являются модальными. В модальное окно можно загрузить любой документ.

Для создания модального окна используется метод *showModalDialog()*. Так же, как и метод *open()*, он принимает в качестве параметров адрес документа (файла), имя окна и строку свойств. Однако формат этой строки другой. В частности, параметры в строке разделяются точкой с запятой, размеры окна и координаты его верхнего левого угла требуют указания единиц измерения (например, px – пикселы). Кроме того, этот метод не возвращает ссылку на объект окна, поскольку она не нужна для модального окна.

В табл. 4.2 приведен список свойств окна, созданного методом *showModalDialog()*, передаваемых в строке features.

Таблица 4.2

Свойства окна, созданного методом showModalDialog()

Свойство	Значения	Описание
order	thick, thin	Размер рамки вокруг окна (толстая/тонкая)
center	yes, no (1, 0)	Выравнивание окна по центру главного
dialogHeight	Число + единицы измерения	Высота окна
dilalogLeft	Число + единицы измерения	Горизонтальная координата
dialogTop	Число + единицы измерения	Вертикальная координата
dialogWidth	Число + единицы измерения	Ширина окна
font	Строка таблицы стилей	Стиль, определенный по умолчанию для окна
font-family	Строка таблицы стилей	Вид шрифта, определенный по умолчанию для окна
font-size	Строка таблицы стилей	Размер шрифта, определенный по умолчанию для окна
font-style	Строка таблицы стилей	Тип шрифта, определенный по умолчанию для окна
font-variant	Строка таблицы стилей	Вариант шрифта (обычный/курсив), определенный по умолчанию для окна

Окончание табл. 4.2

Свойство	Значения	Описание
font-weight	Строка таблицы стилей	Толщина шрифта, определенная по умолчанию для окна
help	yes, no, 1, 0	Включение кнопки Help в верхнюю панель
maximize	yes, no, 1, 0	Включение кнопки Maximize в верхнюю панель
minimize	yes, no, 1, 0	Включение кнопки Minimize в верхнюю панель

§ 4.2. Фреймы

Фрейм представляет собой прямоугольную область окна браузера, в которую можно загрузить HTML-документ. Разбиение окна на фреймы происходит с помощью HTML-кода, содержащегося в отдельном HTML-файле, который называется установочным. Установочный файл содержит только теги разметки фреймов, то есть их относительное расположение, размеры, ссылки на загружаемые в них документы и другие параметры фреймов. В нем не должны присутствовать теги других элементов и текстовая информация, за исключением разве что тегов *<META>*. В установочном файле можно написать сценарий, создающий фреймовую структуру документа. Напомним, что разбиение окна браузера на несколько фреймов обычно производится с помощью тега *<FRAMESET>*. Внутрь этого тега вставляются теги *<FRAME>* с атрибутами, указывающими имя фрейма и адрес HTML-документа, который будет отображаться в этом фрейме. В следующем примере создается два фрейма, расположенные друг над другом:

```
<HTML>
<FRAMESET ROWS="30%,70%"
<FRAME SRC="документ1" NAME="frame1">
<FRAME SRC="документ2" NAME="frame1">
</FRAMESET>
</HTML>
```

Результат выполнения кода представлен на рис. 4.1.

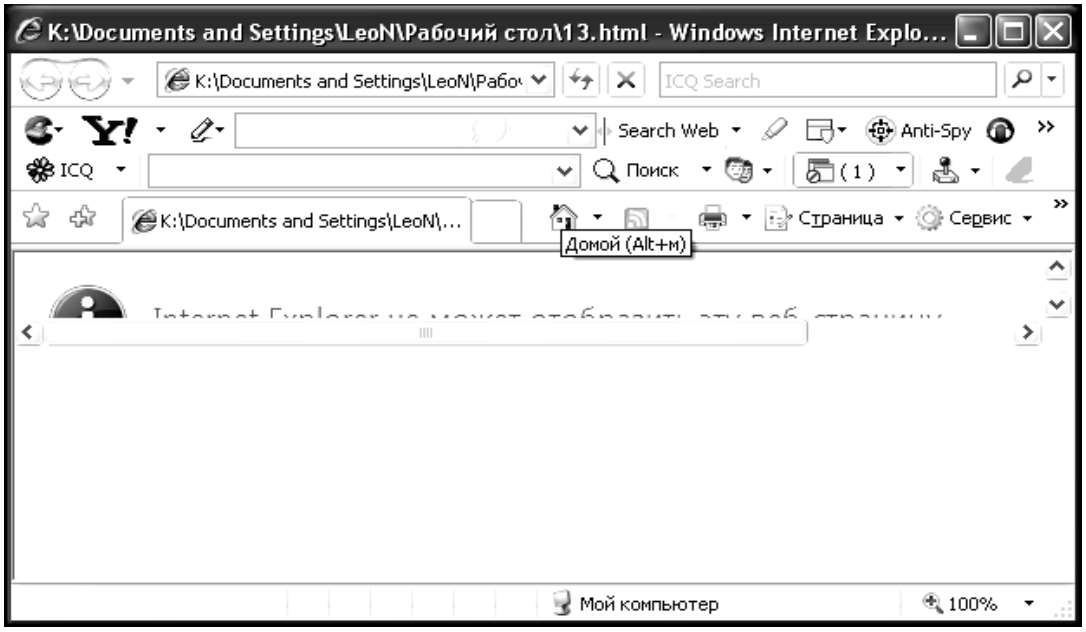

Рис. 4.1. Пример выполнения кода

Это так называемое вертикальное расположение фреймов. Если вместо атрибута *ROWS* в теге <*FRAMESET*> использовать атрибут *COLS*, то фреймы будут расположены горизонтально: второй фрейм находится справа от первого. Используя вложение тега <*FRAMESET*>, можно разбить уже имеющийся фрейм на другие два фрейма и т. д.

4.2.1. Отношения между фреймами и главным окном

При разбиении окна на два фрейма в объектной модели браузера возникает иерархия объектов, в которой главное окно браузера представляется в виде главного, родительского фрейма, а созданные два фрейма являются дочерними фреймами, или, иначе, фреймами-потомками. При разбиении любого из этих фреймов на следующие два фрейма последние являются дочерними для исходного. Итак, между фреймами возникает отношение родители – потомки (родители – дочки). При запуске браузера объект главного окна window формируется автоматически. Если в это окно загружается документ с фреймовой структурой, то главное окно получает статус родительского фрейма по отношению ко всем остальным. С другой стороны, каждый тег <*FRAME*> внутри контейнера <*FRAMESET*> создает свой собственный объект window, в который загружается соответствующий документ. Каждому из фрей-

мов соответствует свой объект *document*. С точки зрения объекта document ему соответствует единственный контейнер – фрейм. Хотя родительский объект и не виден пользователю, он все равно присутствует в объектной модели.

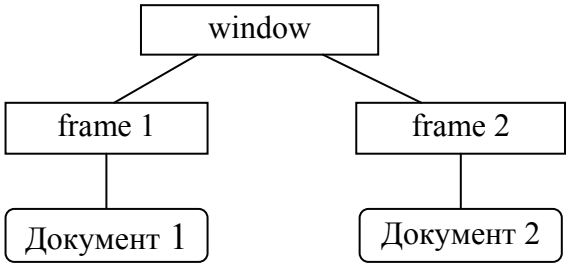

Рис. 4.2. Родительское окно и два фрейма

Итак, в вершине иерархии находится окно браузера window. Оно поделено на два фрейма с именами, например frame1 и frame2. Объект window является родительским по отношению к дочерним frame1 и frame2 (рис. 4.2). Для ссылки на родительское окно используется ключевое слово parent. Допустим, пользователь активизирует некоторую ссылку в первом фрейме, но соответствующий документ должен загрузиться не в этот же фрейм, а в другой. Для решения этой задачи необходимо рассмотреть три случая:

- главное окно получает доступ к фрейму-потомку;
- фрейм-потомок получает доступ к родительскому (главному) окну;
- фрейм-потомок получает доступ к другому фрейму-потомку.

Между объектом window и объектами frame1 и frame2 существует прямая связь родитель – потомок. Поэтому, если вы пишете сценарий в установочном файле, создающем эти фреймы, то к фреймам можно обращаться по имени, как это показано на рис. 4.3.

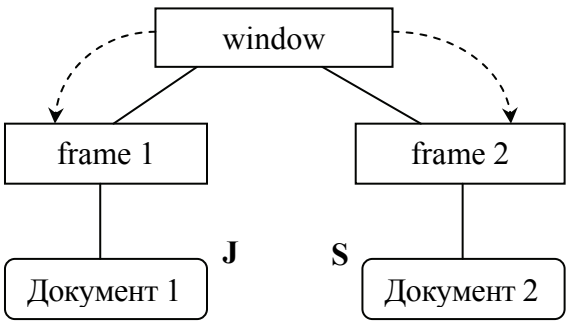

Рис. 4.3. Обращение к фреймам по имени

Иногда может потребоваться получить доступ к родительскому окну. Например, это бывает необходимо, если вы хотите при следующем переходе убрать все фреймы. Удаление фреймов означает лишь загрузку нового документа вместо содержащего фреймы – в рассматриваемом случае загрузку документа в родительское окно. Это можно сделать с помощью доступа к родительскому окну на основе свойства parent (рис. 4.4). Чтобы загрузить новый документ, следует использовать объект *location* из родительского окна (заметим, что каждый фрейм имеет собственный объект *location*): требуется внести в *location.href* родительского окна новый URL-адрес. Таким образом, чтобы загрузить новый документ в родительское окно, требуется записать в сценарии следующее:

parent.location.href="URL-адрес_документа"

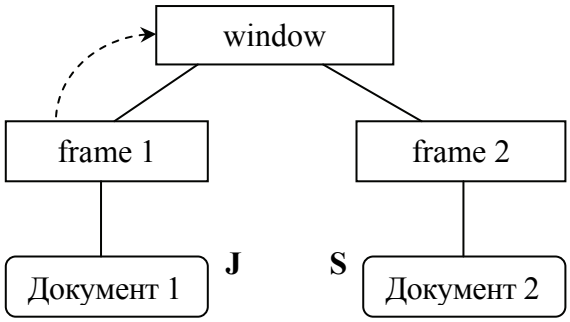

Рис. 4.4. Доступ к родительскому окну на основе свойства parent

Наконец, довольно часто необходимо решать задачу обеспечения доступа из одного фрейма-потомка к другому (рис. 4.5). Например, находясь в одном фрейме-потомке, можно записать что-нибудь в другой фрейм-потомок. Однако между фреймами-потомками не существует прямой связи. Поэтому мы не можем просто вызвать фрейм frame2 по имени, находясь во фрейме frame1. Фрейм frame1 ничего не знает о frame2, но frame2 существует с точки зрения родительского окна. По этой причине необходимо сначала обратиться к родительскому окну, затем – к frame2, и лишь потом к объекту document, который разместился во втором фрейме:

parent. frame2 .document. write("Привет от первого фрейма!")

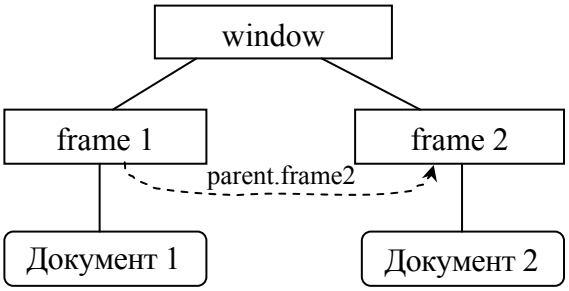

Рис. 4.5. Обеспечение доступа
из одного фрейма-потомка к другому

Рассмотрим, как можно изменить элемент в одном фрейме из другого. При щелчке на тексте в правом фрейме в левом изменяется один из текстовых элементов. Ниже показаны HTML-коды соответственно для левого и правого фреймов, а также порождаемый ими внешний вид страницы. Документ в левом фрейме с именем LEFT:

```
<HTML>
Один<B1>
Два
<H1 ID = "XXX">Три</H1>
</HTML>
Документ в правом фрейме:
<HTML>
<SCRIPT>
function change() {
parent.LEFT.document.all.XXX.innerText = "Ура!"
I
</SCRIPT>
<H1 onclick = "change() ">Щелкни здесь</H1>
</HTML>
```

В теле функции *change()*, которая делает нужные изменения, происходит обращение к левому фрейму с именем LEFT (задается в установочном HTML-файле) через parent. Далее document.all.XXX обеспечивает доступ к элементу с идентификатором "XXX" (в примере это заголовок 1-го уровня). Здесь all – коллекция всех элементов документа. Собственно изменение элемента происходит за счет присвоения значения свойству *innerText* (в примере это слово *"Ура!"*).

Обратите внимание, что изменения в одном фрейме по событию в другом происходят без перезагрузки HTML-документа. Для изменения элементов можно использовать, кроме *innerText*,

свойства *outerText*, *innerHTML* и *outerHTML*. Выбор свойства зависит от того, что именно и насколько вы хотите изменить.

Описанным выше способом можно организовать, например, такой сценарий: щелчок на миниатюре (маленьком изображении) в одном фрейме выводит в нем же полномасштабное изображение, а в другом – краткое описание.

Фреймы удобно использовать при создании навигационных панелей. В одном фрейме располагаются ссылки, а второй предназначен для отображения документов, вызываемых при активизации соответствующих ссылок. При активизации ссылки документ загружается не в тот же фрейм, где находятся ссылки, а в другой.

Пример

Навигационная панель. Окно браузера разделено на два фрейма: первый выполняет роль навигационной панели, а второй – окна для отображения документов.

frames.htm – установочный файл.

```
<HTML>
<FRAMESET COLS = "25%,75%">
<FRAME SRC = " menu.htm " NAME = "menu">
<FRAME SRC = " start.htm " NAME = " main " >
</FRAMESET>
</HTML>
```

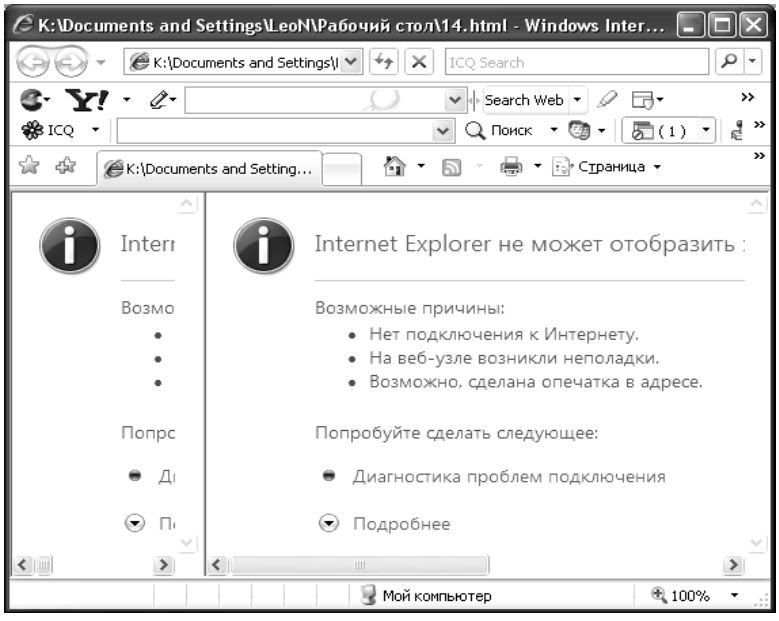

Рис. 4.6. Пример выполнения кода

Результат выполнения кода представлен на рис. 4.6.

Здесь start.htm – документ, который будет первоначально показан во фрейме main.

menu.htm – навигационная панель.

```
<HTML>
<SCRIPT >
function load (url){
parent.main.location.href = url ;
}
</SCRIPT>
<BODY>
<A HREF = "javascript : load( ' первый . htm') ">Первый</A>
    <A HREF = "второй. htm" TARGET = "main">Второй </A>
<A HREF = "третий.htm" TARGET = "Top">Третий </A>
</BODY>
</HTML>
```

Результат выполнения кода представлен на рис. 4.7.

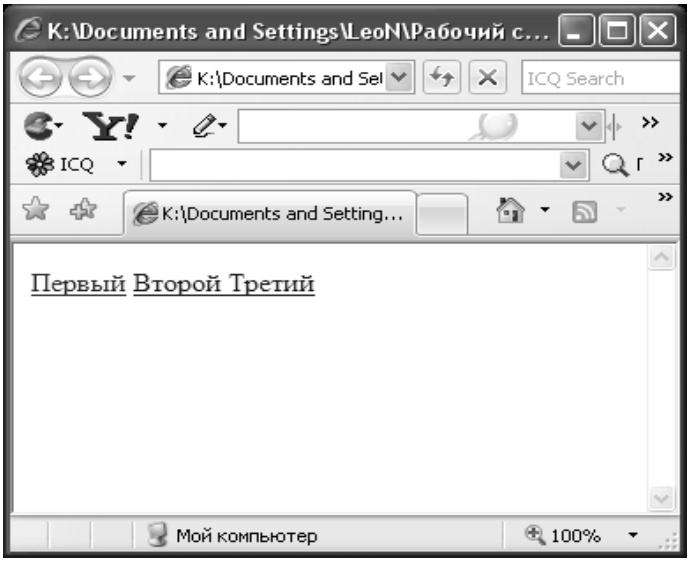

Рис. 4.7. Пример выполнения кода

Здесь показано несколько способов загрузки новой страницы во фрейм main. В первой ссылке используется функция *load()*. Вместо атрибута *TARGET* указание на фрейм выполняет функция. Функции *load()* в качестве параметра передается строка 'первый.htm', указывающая, какой файл следует загрузить. При этом место, куда он

будет загружен, определяется самой функцией *load()*. Во второй ссылке используется атрибут *TARGET*. Третья ссылка показывает, как можно избавиться от фреймов. Чтобы удалить фреймы с помощью функции *load()*, достаточно написать в ней следующую строку:

parent.location.href = url

Атрибут *TARGET* в теге ссылки *<A HREF>* обычно применяется в тех случаях, когда требуется загрузить одну страницу в один фрейм. Язык сценариев используют, если необходимо при активизации ссылки выполнить несколько действий, например, загрузить несколько страниц в разные фреймы. Для ссылок из родительского окна к объектам его дочерних фреймов можно воспользоваться коллекцией frames всех фреймов. Коллекция фреймов представляет собой массив объектов фреймов. Обратиться к конкретному фрейму из этой коллекции можно по индексу или по имени фрейма, указанному в качестве значения атрибута *NAME* в теге *FRAME*:

window.frames[индекс]
window.имя_фрейма

Заметим, что индекс 0 соответствует первому дочернему фрейму в порядке, определенном следованием тегов *<FRAME>* в контейнере *<FRAMESET>*.

Приведенные выше шаблоны ссылок на фреймы из родительского окна используются как префиксы в полных ссылках к объектам, содержащимся во фреймах. При этом следует помнить, что если нас интересуют объекты документа, загруженного во фрейм, то прежде чем обратиться к ним, следует упомянуть объект document. Например

window.frames(0).document.all.My input.value
window.LEFT.document.all.My input.value

Ссылка из дочернего фрейма непосредственно на родительский производится с помощью ключевого слова parent. Если имеется еще один фрейм более высокого уровня, то ссылка на него выглядит так: parent.parent. Аналогичным образом можно построить ссылку из дочернего фрейма до прапрадедушки. Чтобы сразу обратиться к родительскому окну, находящемуся на вершине иерархии, можно использовать ключевое слово top.

При использовании ссылки top следует учитывать то обстоятельство, что ваш сайт может быть загружен в один из фреймов другого сайта. В этом случае указанный вами объект top окажется совсем другим, и ссылки, построенные с его использованием, не будут правильно работать. Поэтому рекомендуется использовать parent для ссылок на вышестоящий фрейм (окно).

4.2.2. Предотвращение использования фреймов

Ссылки top и self можно использовать для предотвращения отображения вашего сайта внутри фреймов другого сайта. Для этого необходимо, чтобы документ верхнего уровня проверял, в какое окно он загружен. Это должно быть самое верхнее (top) или родительское (parent) окно. Если это действительно так, ссылка на свойство top совпадает со ссылкой self на текущее окно. При несовпадении этих значений документ следует перезагрузить заново, но уже в окно верхнего уровня. Сценарий, выполняющий эту работу, необходимо разместить в начале документа. Вот его код:

```
<SCRIPT>
if (top != self)
top.location = location
</SCRIPT>
```

4.2.3. Проверка загрузки фреймов

При посещении вашего многофреймового сайта пользователь может сделать закладку (поместить в папку «Избранное») только на один фрейм, в то время как все средства навигации по сайту находятся в другом фрейме. При следующем посещении в браузер загрузится усеченный вариант вашего сайта. При этом пострадает как пользователь, так и ваша репутация.

Следующий простой сценарий проверяет, загружается ли вебстраница в своем наборе фреймов, сравнивая URL-адреса окна top и текущего окна. Если они совпадают, то необходимо загрузить набор фреймов, определяемый в файле frameset.htm.

```
<SCRIPT>
if (top.location.href == window.location.href)
top.location.href = "frameset.htm"
</SCRIPT>
```

4.2.4. Плавающие фреймы

Для вставки одного HTML-документа в тело другого средствами браузера пользователя, а не сервера, служит контейнерный тег *<IFRAME>*:

```
<IFRAME SRC = "адрес_документа"></IFRAME>
```

Элемент, задаваемый этим тегом, можно позиционировать с помощью параметров таблицы стилей (тег *<STYLE>* или атрибут *STYLE*). Если его положение не указано явно, то он позиционируется в соответствии с положением *<IFRAME>* в HTML-коде. Внешне этот элемент выглядит как прямоугольная область (с полосами прокрутки или без них), в которой отображается документ из некоторого HTML-файла. Такие окна иногда называют плавающими фреймами. HTML-документы, загружаемые в плавающие фреймы, могут иметь сценарии и прочие средства, присущие любому HTML-документу. Ниже приведен пример загрузки трех различных HTML-файлов в плавающие фреймы.

```
<HTML>
<H3>Включение HTML-документов на стороне клиента</H3>
    <H4>Использование тега <iframe></H4>
Здесь в трех окнах показаны страницы моего сайта<  >
<IFRAME SRC = "examples.htm" ></IFRAME>
<IFRAME SRC = " flashx.htm " ></IFRAME>
<p>
<IFRAME SRC = "i_is.htm" STYLE = " position: absolute ; top: 310; width: 500;
height: 250" SCROLLING = no" ></IFRAME >
</HTML>
```

Плавающий фрейм ведет себя почти так же, как и обычный фрейм. Если он создан, то его можно найти в коллекции frames.

Документ с тремя документами из внешних файлов, загруженными в плавающие фреймы, показан на рис. 4.8.

Среди его свойств рассмотрим свойство *align* – выравнивание плавающего фрейма относительно окружающего содержимого документа.

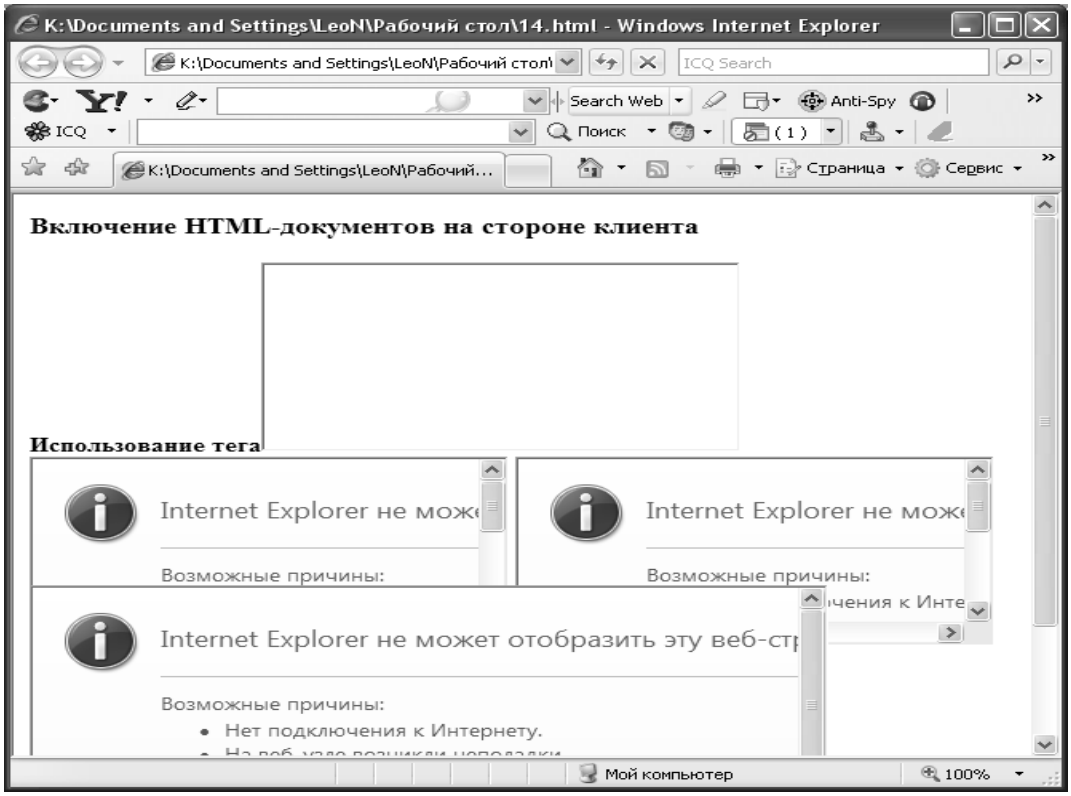

Рис. 4.8. Документ с тремя документами из внешних файлов, загруженными в плавающие фреймы

Возможные значения *align*:

- absbottom – выравнивает нижнюю границу фрейма по подстрочной линии символов окружающего текста;
- absmiddle – выравнивает середину фрейма по центральной линии между top и absbottom окружающего текста;
- baseline – выравнивает нижнюю границу фрейма по базовой линии окружающего текста;
- bottom – совпадает с baseline (только в IE);
- left – выравнивает фрейм по левому краю элемента-контейнера;
- middle – выравнивает воображаемую центральную линию окружающего текста по воображаемой центральной линии фрейма;
- right – выравнивает фрейм по правому краю элемента-контейнера;
- texttop – выравнивает верхнюю границу фрейма по надстрочной линии символов окружающего текста;
- top – выравнивает верхнюю границу фрейма по верхней границе окружающего текста.

§ 4.3. Всплывающие окна

Всплывающее окно не имеет органов управления и располагается над документом, в котором оно было создано, в том числе и над диалоговыми окнами. Щелчок кнопкой мыши где-либо вне этого окна или вызов другого приложения приводит к его закрытию. Оно также не отображается на панели задач Windows. После создания всплывающего окна его заполняют содержимым. При этом необходимо следить, чтобы содержимое вписывалось в заданные границы окна, поскольку оно не имеет полос прокрутки. Такие окна удобно использовать в качестве больших всплывающих подсказок для вывода контекстной справочной информации.

Всплывающим окнам соответствует объект popup. Он создается только в сценарии для браузера IE5.5+ с помощью метода *window.createPopup()*.

Свойства объекта popup:

• document – свойство, имеющее в качестве значения ссылку на документ, содержащийся во всплывающем окне; через него можно задать ряд параметров, как самого окна, так и его содержимого:

```
mypopup. document .body. style . border = "solid 4px black" // границы окна
mypopup.document. body. style . background = "yellow"  // цвет фона окна
mypopup.document.body.style.color = "blue"          /* цвет текста в окне */
```

• isOpen – пока всплывающее окно отображается на экране, это свойство имеет значение true, в противном случае – false; применяется в выражениях сценария, которые выполняются при уже открытом окне.

Методы объекта popup:

• show(left, top, width, height [позиционирование]) – отображает всплывающее окно после создания объекта popup с помощью метода window.createPopup() и заполнения его содержимым. Если окно исчезло из-за того, что пользователь щелкнул где-то в поле браузера, то для его повторного открытия следует заново выполнить метод show(). Первые четыре параметра метода задают координатное позиционирование и размеры окна. Параметры left и top определяют координаты верхнего левого угла окна относительно экрана мо-

нитора, а не окна браузера. Последний необязательный параметр позволяет задать другое координатное пространство как ссылку на элемент HTML-документа. Например, указав document.body, вы ограничиваете координатное пространство окном браузера.

• hide() – позволяет скрыть созданное и отображаемое всплывающее окно.

Заполнение всплывающего окна содержимым производится с помощью свойства innerHTML, принимающего в качестве значения строку, содержащую теги HTML. Таким образом, в окно можно вставлять не только тексты, но и изображения и другие элементы.

Пример

mypopup.document.body.innerHTML = "<IMG SRC = ' picture.jpg ' > "

Ниже приводится пример HTML-документа с двумя кнопками, щелчок на которых вызывает одну и ту же функцию pop(xcontent), открывающую всплывающее окно и заполняющую его содержимым, которое задано параметром xcontent:

```
<HTML>
<H3>Vsplivaus4ie okna</H3>
<BUTTON onclick = "pop('Hello') ">Prosto tekst </BUTTON>
<BR><BR>
<BUTTON onclick = "pop('<IMG SRC = K:\Documents and Settings\LeoN\Мои документы\Мои рисунки\350.jpg>Leopardy ') ">Kartinka s tekstom</BUTTON>
<SCRIPT>
var mypopup = window. createPopup()
var popupBody = mypopup.document.body
popupBody.style.border = "solid 2px green"
popupBody.style.padding = "5px"
popupBody.style.color = "blue"
popupBody.style.background = "ffffd0"
function pop(xcontent) {
popupBody.innerHTML = "<p>"+ xcontent + "</p>"
mypopup.show(130,90,400,200,document.body)
}
</SCRIPT>
</HTML>
```

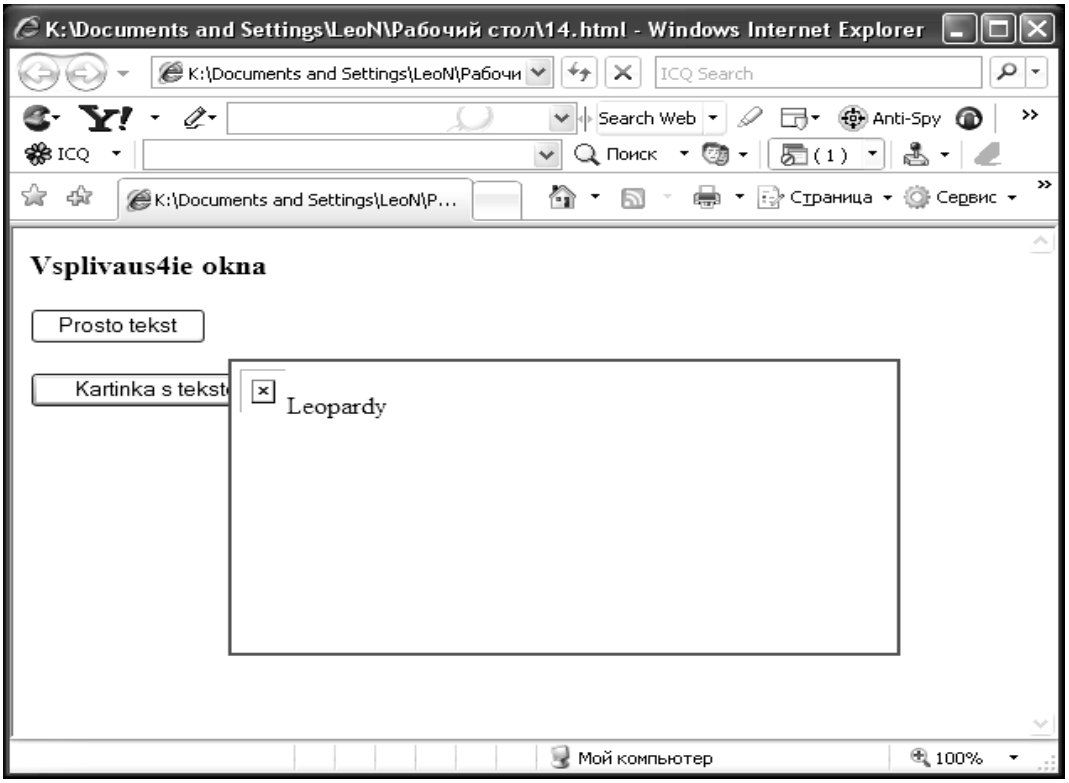

Рис. 4.9. Пример HTML-документа с двумя кнопками

Пример всплывающего окна с изображением и текстом показан на рис. 4.9.

§ 4.4. Динамическое изменение элементов документа

HTML-документ, загруженный в браузер, можно изменять с помощью сценариев. Поскольку речь идет не о файле на диске, а о его образе в браузере в оперативной памяти компьютера, то говорят о динамическом изменении документа. Существует три основных способа динамических изменений:

- с помощью метода *write ()*;
- путем изменения значений свойств, соответствующих атрибутам HTML-тегов и параметрам каскадных таблиц стилей;
- путем изменения значений свойств *innerText*, *outerText*, *innerHTML*, *outerHTML*, которые имеют почти все объекты, заданные с помощью тегов.

§ 4.5. Использование метода write()

Метод *write()* объекта *document* принимает в качестве параметра строку, содержащую HTML-код и/или просто текст. Выполнение в сценарии выражения *document.write(строка)* приводит к дописыванию в текущий HTML-документ содержимого параметра *строка* и немедленной его интерпретации браузером. В результате документ и его объектная модель обновляются. При этом файл с исходным HTML-кодом остается без изменений. Если требуется полностью заменить текущий документ, то сначала применяют метод очистки документа *document.clear()*, а затем *document.write(строка)*. Однако при такой кардинальной трансформации текущего документа следует быть осторожным. Наиболее безопасный прием – сначала сгенерировать содержимое нового HTML-документа с помощью сценария в текущем документе, а затем отправить HTML-код в новое окно или в другой фрейм многофреймового документа. Если быть более точным, то следует отметить, что метод *write()* может принимать произвольное количество строковых параметров:

document. write (строка1, [, строка2 . . .[, строкаN]])

Здесь квадратные скобки указывают лишь на необязательность заключенных в них параметров. Если указывается несколько параметров, то они разделяются запятыми. Заметим также, что весь HTML-код документа можно записать как одну строку.

Кроме метода *write()* можно использовать для тех же целей и метод *writeln()*, имеющий такой же синтаксис.

§ 4.6. Изменение значений атрибутов элементов

Элементы HTML-документа, как известно, задаются тегами, большинство из которых имеют атрибуты (параметры). В объектной модели документа тегам элементов соответствуют объекты, а атрибутам – свойства этих объектов. В большинстве случаев названия свойств объектов совпадают с названиями атрибутов, но, в отличие от последних, записываются в нижнем регистре. Это же относится и к параметрам таблиц стилей. Однако это общее правило

может иметь исключения. Поэтому им нужно пользоваться следующим образом: ищите в справочнике свойство, похожее на атрибут, и обращайте внимание на его написание. Например, тегу графического изображения *<IMG ID = "myimg" SRC = "picture.jpg">* соответствует объект document.all.myimg, а атрибуту SRC – свойство *document.allmyimg.src*, значением которого является имя (URL-адрес) файла с изображением. С помощью сценария можно присвоить этому свойству новое значение, и в HTML-документе произойдет замена графического элемента.

Многие параметры элементов задаются с помощью таблиц стилей, например посредством атрибута *STYLE*. Так, для позиционирования изображения можно использовать следующий HTML-код:

```
<IMG ID = "myimg" SRC = " picture.jpg "
STYLE = " position: absolute ; top:20; left : 50: z-index:3">
```

Чтобы изменить в сценарии параметры стиля элемента, следует присвоить новые значения соответствующим свойствам объекта *style*:

```
document.all.myimg.style.top = 30
document. all . myimg. style . top = 100
document.all.myimg.style.zindex = -2
```

Параметр z-index в таблице стилей, указывающий слой (относительное положение выше-ниже) для элемента, в объектной модели представляется свойством *zindex.*

Между обозначениями в HTML и таблицах стилей, с одной стороны, и обозначениями соответствующих свойств объектов, с другой стороны, много общего, но имеются и различия.

§ 4.7. Изменение элементов

Теги элементов документа могут содержать текст и/или другие теги. С помощью перечисленных выше свойств можно получить доступ к содержимому элемента. Изменяя значения этих свойств, можно изменить сам элемент, частично или полностью. Например, можно заменить только надпись на кнопке, а можно превратить кнопку в изображение или Flash-анимацию.

Значением свойства *innerText* является все текстовое содержимое, заключенное между открывающим и закрывающим тегами элемента. Если в эту строку входят HTML-теги, то они игнорируются. Обратите внимание, что в значение свойства *innerText* данные открывающего и закрывающего тегов соответствующего элемента не входят. Таким образом, *innerText* следует понимать как весь внутренний текст, содержащийся в контейнере.

Свойство *outerText* аналогично свойству *innerText*, но отличается от него тем, что включает в себя и данные, содержащиеся в открывающем и закрывающем тегах элемента. Таким образом, *outerText* следует понимать как весь текст, содержащийся в контейнере, включая и его внешние теги.

При присвоении свойствам *innerText* и *outerText* новых значений следует иметь в виду, что если значение содержит HTML-теги, то они не интерпретируются, а выводятся на экран просто как текст. Так, для приведенного выше фрагмента HTML-кода выполнение в сценарии выражения

document.all. my. innerText = "<BUTTON>Щелкни здесь </BUTTON>"

не создаст в документе кнопку, а лишь выведет строку с текстом, указанным справа от оператора присвоения.

Рассмотренные выше свойства *innerText* и *outerText* не столь эффективны, как замечательные свойства *innerHTML* и *outerHTML*.

Свойство *innerHTML* для любого элемента имеет в качестве значения строку, содержащую HTML-код, заключенный между открывающим и закрывающим тегами элемента. Иначе говоря, *innerHTML* содержит внутренний HTML-код контейнера элемента. Присвоение этому свойству нового значения, содержащего HTML-код, приводит к интерпретации этого кода. Разумеется, новое значение может и не содержать тегов.

Свойство *outerHTML* аналогично свойству innerHTML, но отличается тем, что содержит весь HTML-код, включая внешние открывающий и закрывающий теги элемента.

Если в сценарии выполнить, например, выражение

document.all.my.innerHTML = "<BUTTON>Щелкни здесь</BUTTON>",

то ссылка, изображение и текст будут заменены кнопкой с надписью Щелкни здесь. При этом контейнерный тег *<DIV ID = "my">* сохранится. Если же вместо свойства *innerHTML* использовать свойство *outerHTML*, то кнопка также появится, но уже без контейнера *<DIV ID = "my">*.

Заметим, что свойства *innerHTML* и *outerHTML* могут применяться и к элементам, которые задаются неконтейнерными тегами, например тегом *<IMG>*. В этом случае значения *innerHTML* и *outerHTML* всегда совпадают.

§ 4.8. Загрузка изображений

Большинство веб-страниц содержат графические элементы, которые используются не только для украшения страниц, но и в качестве информационного наполнения: иллюстрированные каталоги товаров, схемы, чертежи, географические карты, фотогалереи и т. п. Если файлы с графикой велики и/или их много, то загрузка такой страницы в браузер может потребовать слишком много времени. Для ускорения загрузки используют специальные приемы. Так, нередко сначала загружают изображения с низким разрешением из небольших по объему файлов. На их основе создаются ссылки на файлы с графикой полноценного разрешения, которые загружаются только при щелчке на ссылке. Можно также использовать в теге *<IMG>* кроме атрибута *SRC* атрибут *LOWSRC*, позволяющий загрузить сначала изображение с низким разрешением, а затем, по мере приема, заменить ее рисунком с большим разрешением. Здесь мы рассмотрим способ ускорения загрузки графики, предоставляемый JavaScript. С помощью сценария можно организовать предварительную загрузку изображений в кэш-память браузера, не отображая их на экране. Это особенно эффективно при начальной загрузке страницы. Пока изображения загружаются в память, оставаясь невидимыми, пользователь может рассматривать текстовую информацию, его не раздражает медленное появление графических элементов. Для предварительной загрузки изображения следует создать его объект в памяти браузера. Такой объект несколько отличается от объекта изображения, создаваемого с помощью тега *<IMG>*. Как и все объекты, создаваемые сценариями, объект изображения не отображается в окне

браузера. Однако его наличие в коде документа уже обеспечивает загрузку самого изображения при загрузке документа. Чтобы создать в памяти объект изображения, необходимо в сценарии выполнить следующее выражение:

myimg = new image(ширина, высота)

Параметры функции-конструктора объекта определяют размеры изображения и должны соответствовать значениям атрибутов *WIDTH* и *HEIGHT* тега *<IMG>*, который используется для отображения предварительно загруженного изображения.

Для созданного в памяти объекта изображения myimg можно задать имя или, в общем случае, URL-адрес графического файла. Это делается с помощью свойства src:

myimg. src = "URL-адрес_изображения"

Данное выражение предписывает браузеру загрузить в кэш-память указанное изображение, но не отображать его. После загрузки в кэш-память всех изображений и загрузки всего документа можно сделать их видимыми. Для этого нужно свойству src элемента *<IMG>* присвоить значение этого же свойства объекта изображения в кэш-памяти. Например:

document.images[0].src = myimg.src

Здесь слева от оператора присвоения указано свойство src первого в документе элемента, соответствующего тегу *<IMG>*, а справа – свойство *src* объекта изображения в кэш-памяти.

§ 4.9. Управление процессами во времени

Можно периодически, через заданный интервал времени, запускать код (например, функцию) JavaScript. При этом создается эффект одновременного (параллельного) выполнения вычислительных процессов. Например, можно запустить несколько функций, перемещающих на экране различные видимые объекты. Эти объекты будут двигаться одновременно. Иногда требуется организовать временную задержку перед выполнением какой-то функции, чтобы ранее начатый процесс успел завершиться.

Все это относится к задачам управления вычислительными процессами во времени. Для организации постоянного периодического (через заданный интервал времени) выполнения некоторого выражения или функции служит метод *setInterval()* объекта *window*. Этот метод имеет следующий синтаксис:

setInterval (выражение, период [, язык])

Первый параметр представляет собой строку, содержащую выражение (в частности, вызов функции). Второй параметр – целое число, указывающее временную задержку в миллисекундах перед последующими выполнениями выражения, указанного в первом параметре. Третий, необязательный параметр указывает язык, на котором написано выражение; по умолчанию – JavaScript. Метод *setInterval()* возвращает некоторое целое число – идентификатор временного интервала, который может быть использован в дальнейшем для прекращения выполнения процесса, запущенного с помощью данного метода (см. ниже метод *clearInterval()*).

Пусть, например, требуется, чтобы некоторая функция *myfunc()* выполнялась периодически через 0,5 с. Тогда в сценарии следует записать следующее выражение:

```
setInterval("myfunc()", 500)
```

Тот факт, что первый параметр метода *setInterval()* является строкой, обусловливает некоторые особенности передачи параметров периодически вызываемой функции. Если периодически вызываемая функция принимает параметры, то необходимо сначала сформировать строку, содержащую имя этой функции, круглые скобки, значения параметров и запятые между ними, а затем передать ее в качестве первого параметра методу *setInterval()*. В следующем примере показано, как передать методу *setInterval()* функцию с двумя параметрами, param1 и param2, значения которых определены в другом месте сценария:

```
van xstr = "myfunc(" + param1 +" , "+ param2 +")"
setInterval(xstr , 500)
```

Выражение, переданное методу *setInterval()*, будет периодически выполняться сколь угодно долго. Если это выражение осуществляет, например, приращение координат какого-нибудь видимого элемента документа, то этот элемент будет перемещаться в окне браузера.

Для остановки запущенного временного процесса служит метод clearInterva1(идентификатор), который принимает в качестве параметра целочисленный идентификатор, возвращаемый соответствующим методом *setInterval()*, например:

```
var myproc = setInterval("myfunc () , 100")
if (confirm("Прервать процесс ?"))
clearInterval(myproc)
```

Чтобы выполнить выражение с некоторой временной задержкой, используется метод *setTimeout()*. Этот метод объекта *window* имеет следующий синтаксис:

```
setTimeout(выражение, задержка [, язык])
```

Первый параметр представляет собой строку, содержащую выражение (в частности, вызов функции). Второй параметр – целое число, указывающее временную задержку в миллисекундах выполнения выражения, указанного в первом параметре. Третий, необязательный параметр указывает язык, на котором написано выражение; по умолчанию – JavaScript. Метод setTimeout() возвращает некоторое целое число – идентификатор временного интервала, который может быть использован в дальнейшем для отмены задержки выполнения процесса, запущенного с помощью данного метода (см. ниже метод clearTimeout()).

Помните, что это выражение не задерживает выполнение всех последующих выражений сценария. Оно лишь задерживает выполнение функции *myfunc()*.

Для отмены задержки процесса, запущенного с помощью метода setTimeout(), используется метод *clearInterval(идентификатор)*, который принимает в качестве параметра целочисленный идентификатор, возвращаемый соответствующим методом *setTimeout*.

В следующем HTML-документе имеются две кнопки. Щелчок на кнопке Пуск открывает через 5 с новое окно и загружает в него документ mypage.htm. Однако это действие можно отменить с помощью кнопки Отмена, если щелкнуть на ней, пока окно еще не открыто:

```
<HTML>
<BUTTON ID="start">Пуск</BUTTON>
<BUTTON ID="stop">Отмена</BUTTON>
<SCRIPT>
```

```
var myproc
functionstart.onclick(){
myproc = setTimeout ("window.open('mypage.htm')" , 5000)
}
function stop . onclick(){
clearTimeout(myproc)
}
</SCRIPT>
</HTML>
```

Результат выполнения кода представлен на рис. 4.10.

Рис. 4.10. Пример выполнения кода

При использовании методов *setInterval()* и *setTimeout()* следует иметь в виду, что их вторые параметры задают лишь приблизительные значения временных задержек.

§ 4.10. Работа с Cookie

Для хранения небольших объемов информации на диске компьютера пользователя в браузере предусмотрен так называемый механизм cookie. Обычно он используется для хранения имени пользователя и пароля, который вводится в поле формы защищенного веб-сайта, а также информации о предыдущем посещении

сайта. Например, можно сохранить на диске дату последнего посещения сайта данным пользователем. При загрузке сайта эта дата сравнивается с некоторой датой, установленной автором сайта в качестве даты обновления. Если вторая (авторская) дата более поздняя, чем первая, то на веб-странице появляется соответствующая отметка, например текст или изображение с надписью *"New"* или *"Обновлено!"*. Разумеется, сравнение дат и вывод на страницу отметки производится с помощью сценария. По существу, cookie – это единственный способ сохранения данных на диске пользователя, безопасный для него. Как известно, веб-браузеры препятствуют свободному обращению к папкам и файлам на компьютере пользователя. Однако следует помнить, что многие пользователи не любят cookie-записи и всячески их истребляют.

Записи cookie браузер Internet Explorer сохраняет в отдельных текстовых файлах, расположенных в папке Windows\Cookies. Имя такого файла образуется на основе имени пользователя и домена того сервера, на котором создавался cookie файл. Netscape Navigator 4 для Windows создает просто один файл cookie.txt. Вообще говоря, структура данных в cookie-файлах для различных браузеров не столь существенна, поскольку, во-первых, не рекомендуется открывать и изменять эти файлы в текстовых редакторах, а во-вторых, браузеры IE и NN используют одинаковый синтаксис чтения и записи cookie-данных, основанный на использовании свойства *document.cookie*. Итак, данные в cookie-файлах организованы в виде записей. Каждую такую запись можно представить себе в виде строки, содержащей следующие элементы:

- имя записи;
- содержание записи;
- срок хранения (годности) записи;
- домен сервера, который создал запись;
- сведения о необходимости установки безопасного http-соединения для доступа к записям;
- расположение документов, которым разрешен доступ к записям.

Зависимость записей от домена обеспечивает безопасность хранения так называемых невосстанавливаемых паролей (пароль вида имя_пользователя-пароль), поскольку запись, созданную сервером одного домена, не может прочитать сервер с другим доменом. Срок хранения используется браузером для автоматического

удаления просроченных записей, чтобы предотвратить чрезмерное разрастание объема cookie-файлов. Впрочем, IE4+ и NN4+ ограничили объем cookie-файлов 20 записями на каждый домен.

Для записи данных в cookie-файл с помощью JavaScript используется выражение присвоения строки, содержащей cookie-данные, свойству document.cookie. При этом важно соблюдать формат строки:

```
document.cookie = "cookieName=данные
[; expires=Строка_времени_GMT] '
[; path=путь]
[; domain=домен]
[ ; secure] "
```

Здесь квадратные скобки указывают, что заключенное в них содержимое не является обязательным (может быть опущено).

Рассмотрим элементы cookie-строки.

- Имя-данные. Каждая cookie-запись должна иметь имя и строковое значение, которое может быть и пустой строкой. Например, если требуется сохранить слово *"Вася"* в cookie-записи с именем User_Name, то соответствующее выражение JavaScript будет иметь вид:

```
document.cookie = "User_Name=Вася"
```

При выполнении этого выражения браузер пытается найти cookie-запись с таким именем. Если он не находит ее в текущем домене, то создает ее автоматически. Если запись с таким именем уже существует, то браузер заменяет ее данные новыми. Данные в этом элементе cookie-записи не должны содержать точек с запятыми, запятых и пробелов. Чтобы заменить пробелы соответствующими символами (%20), строка с данными предварительно обрабатывается функцией escape().

- Срок хранения. Дата и время хранения (годности) cookie-записи должны быть представлены строкой и содержать данные по Гринвичу (GMT). Например, вычислить дату истечения срока хранения записи месяц спустя после текущей даты можно следующим образом:

```
var expdate = new Date()                    // создаем объект даты
var monthFromNow = expdate . getTime() + (30*24*60*60*1000)
expdate.setTime(monthFromNow)   // устанавливаем значение даты
```

После этого полученную дату следует привести к строковому формату GMT:

document.cookie = "User_Name=Вася; expires = " + expdate. toGMTString()

Cookie-запись можно удалить и до истечения заданного срока хранения, установив новый срок, заведомо уже прошедший:

expdate=Thu. 01-Jan-70 00:00:01 GMT

• path – cookie-записи, производимые компьютером пользователя, имеют путь, принятый по умолчанию (в текущей папке). Однако можно создать копию cookie в другой папке, указав путь к ней в качестве значения этого параметра.

• domain – для синхронизации cookie-данных с определенным документом или группой документов браузер выясняет домен текущего документа и помещает в cookie-файл записи, соответствующие этому домену. Если пользователю требуется просмотреть список всех cookie-записей, содержащихся в свойстве document.cookie, то он должен просмотреть все пары имя-значение, находящиеся в cookie-файле с именем домена текущего документа. Формат представления домена должен охватывать по крайней мере два уровня, например rambler.ru.

• secure – принимает логические значения (true или false).

При создании cookie-записей на стороне клиента (компьютера пользователя) этот параметр опускается.

Теперь займемся чтением и записью данных cookie. Данные cookie, которые можно получить с помощью сценария на JavaScript, представляют собой единственную строку – значение свойства *document.cookie*. Выбор значений отдельных элементов (параметров) cookie производится на основе анализа содержимого этой строки методами объекта *String*. Кроме того, если две и более cookie-записи (до 20) соответствуют одному и тому же домену, то в JavaScript они все равно представляются одной строкой и разграничиваются точкой с запятой и пробелом.

Функция *readCookie(name)* возвращает значение cookie-записи с именем name или null, если такая запись не найдена. В теле этой функции использована еще одна, вспомогательная функция *getCookieVal()*, возвращающая декодированное значение cookie-данных. Декодирование производится с помощью встроенной

функции *unescape()*. Дело в том, что cookie-запись должна представлять собой кодированную строку, полученную путем обработки встроенной функцией *escape()*, чтобы, в частности, заменить пробелы специальными символами (%20).

Функция *writeCookie()* позволяет создать или обновить cookie-запись:

```
function writeCookie(name,value,expires,path,domain,secure)
{
  document.cookie = name + "=" + escape(value) +
  ((expires)  ?  "; expires=" + expires.toGMTString():   "") +
  ((path)  ?  "; path=" + path   :   "")   +
  ((domain)  ?  "; domain="   +  domain :   "")   +
  ((secure)  ?  ";   secure" :   "");
}
```

Эта функция принимает следующие параметры:

- name – строка, содержащая имя cookie-записи (обязательный параметр);
- value – строка, содержащая значение cookie (обязательный параметр);
- expires – объект даты (Date), содержащий срок хранения cookie-записи; если отсутствует, то после завершения работы браузера cookie-запись удаляется;
- path – строка, содержащая путь cookie-записи; если не указан, то используется путь вызванного документа;
- domain – строка, содержащая домен нужной cookie-записи; если не указан, то используется домен вызванного документа;
- secure – логическое значение (true или false), определяющее необходимость использования безопасного HTTP-соединения.

Обратите внимание, что в теле функции *writeCookie()* происходит кодирование значения параметра *value* с помощью встроенной функции *escape()*.

Для удаления cookie-записи можно использовать следующую функцию:

```
function deleteCookie(name, path, domain) {
/* удаление cookie-записи */
if (readCookie(name)) {
```

```
document.cookie =
name + " = " +
((path) ? "; path = " + path : "") +
((domain) ? "; domain="+domain : " " ) +
; expires=Thu, 01-Jan-70 00:00:01 GMT"
```

Эта функция устанавливает дату срока хранения cookie-записи так, что запись будет удалена. Параметр path должен иметь такое же значение, которое использовалось при создании cookie-записи, или иметь пустое значение (null), если при создании записи он не был определен. Таким же образом задается значение параметра domain. Вам рекомендуется поэкспериментировать с чтением, созданием и удалением cookie-записей. При этом следует иметь в виду, что созданные или измененные cookie-записи будут записаны на диск только после закрытия браузера. До этого записи существуют лишь в кэше (оперативной памяти). С другой стороны, cookie-записи загружаются в оперативную память и становятся доступны как значение свойства document.cookie только при запуске браузера.

Можно организовать для зарегистрированных посетителей вашего сайта доступ к специальным страницам, предназначенным только для них. В этом случае сценарий проверяет наличие в cookie-файле записи с некоторым фиксированным именем (например, myspecrecord) и читает ее значение, содержащее пароль. Если пароль правильный, то в документ загружается ссылка на защищенный паролем документ либо сам этот документ. Если такой cookie-записи не нашлось либо пароль в этой записи не верен, то в документ загружается поле ввода пароля. Если введенный пользователем пароль верен, то сценарий создает cookie-запись с именем myspecrecord и записывает в нее пароль. После этого в текущий документ загружается либо ссылка, либо сам защищенный документ. При следующем посещении этого сайта пользователю не придется снова вводить пароль, поскольку сценарий просто считает его из cookie-записи. Это, конечно, справедливо до тех пор, пока cookie-запись с паролем сохраняется на диске компьютера пользователя. Поэтому ее можно обновлять с помощью сценария, указывая новое значение срока хранения, вычисленное на основе текущей даты (например, текущая дата плюс месяц).

Практические задания к главе 4

Задание 1

Самым старым способом показать какой-то документ является всплывающее окно («попап» – от англ. popup window). В современных браузерах – в том случае, если вызов window.open происходит вместе с загрузкой страницы. Браузер хранит специальный внутренний флаг, который говорит – инициировал посетитель выполнение или нет. При клике на кнопку весь код, который выполнится в результате, включая вложенные вызовы, будет иметь флаг «инициировано посетителем» и попапы при этом разрешены, если же код был на странице и выполнился автоматически при ее загрузке – у него этого флага не будет. Попапы будут заблокированы. Здесь есть еще ряд тонких моментов. Например: обработчик onclick вызвал *setTimeout(func, timeout)*. В результате код func выполнится уже после того, как событие onclick обработано. Следует ли передать ему флаг «инициировано посетителем» или нет?

Возможны разные точки зрения, например, IE блокирует такие окна (видимо, проще было реализовать), а Firefox и Chrome/Safari передают флаг, если таймаут меньше секунды. А если больше – то не передают

Вот это окно откроется в Firefox и Chrome (но не в IE):

```
setTimeout(function() {
  window.open("http://ya.ru");
}, 500);
будет заблокировано:
setTimeout(function() {
window.open("http://ya.ru");
}, 2500);
```

Таким образом, современные браузеры расширяют область «действий посетителя» и блокируют попапы не всегда.

Поэкспериментируйте со страницами из прошлых проектов. Объясните, почему один скрипт браузер блокирует, а другой нет.

Задание 2.1

Можно управлять процессом создания окна. Попробуйте задать окно размером 550×400 пикселей, в котором откроется нужный вам сайт.

Предлагаемый вариант

```
<SCRIPT LANGUAGE="javascript">
var newWin = window.open("http://www.bstu.unibel.by/", "displayWindow", "width=600,height=400");
newWin.focus();
newWin.onload = function() {

 // создать div в документе нового окна
 var div = newWin.document.createElement('div');
 div.innerHTML = 'Добро пожаловать!'
 div.style.fontSize = '30px'

 var body = newWin.document.body;
 // вставить первым элементом в новое body
 body.insertBefore(div, body.firstChild);
}
</SCRIPT>
```

Задание 2.2

Перепишите скрипт так, чтобы в окне открывалась какая-либо страница.

Пример

```
<SCRIPT LANGUAGE="javascript">
var newWin = window.open("http://www.bstu.unibel.by/", "displayWindow", "width=600,height=400");
newWin.focus();
newWin.onload = function() {

 // создать div в документе нового окна
 var div = newWin.document.createElement('div');
 div.innerHTML = 'Добро пожаловать!'
 div.style.fontSize = '30px'
 var body = newWin.document.body;
```

```
// вставить первым элементом в новое body
body.insertBefore(div, body.firstChild);

}
</SCRIPT>
```

Задание 3

Написать функцию, которая открывает окно. Документ, который появится в окне, должен иметь зеленый фон и заголовок TITLE: "Привет, "имя пользователя", вот твое окно!" Имя пользователя можно узнать с помощью запроса (prompt). Разумеется, добавьте еще ссылку, которая закроет окно.

Возможное решение

В начале функции добавить команду prompt; разбить команду TITLE на три части и внести в нее переменную name; цвет фона поменять на зеленый.

Вот готовый сценарий:

```
<SCRIPT type="text/javascript">
function openindex()
{
var name=prompt("Как вас зовут?","Напишите здесь")

var OpenWindow=window.open("", "newwin", "height=300,width=300,
status=yes");
OpenWindow.document.write("<HTML>")
OpenWindow.document.write("<TITLE>")
OpenWindow.document.write("Привет, " +name+ "! Вот
ваше окно!")
OpenWindow.document.write("</TITLE>")
OpenWindow.document.write("<BODY BGCOLOR='green'>")
OpenWindow.document.write("<CENTER>")
OpenWindow.document.write("<h2>Новое окно</h2>")
OpenWindow.document.write("<a href="
onClick='self.close()'>Эта ссылка закроет окно</a>")
OpenWindow.document.write("</CENTER>")
OpenWindow.document.write("</BODY>")
OpenWindow.document.write("</HTML>")
```

```
}
</SCRIPT>
  <body bgcolor="xxxxxx" onLoad="openindex()">
```

Задание 4

Сделайте так, чтобы при отмене (Cancel), кроме окна, еще появлялась какая-нибудь надпись в строке состояния.

Можно также попробовать сделать так, чтобы при выборе ОК страница перехода открывалась в новом окне.

Возможное решение

В параграфе заголовка HEAD помещаем следующий код.

```
<SCRIPT LANGUAGE="javascript">
function go()
{
if (confirm("Хотите на INTUIT?") )
{
parent.location='http://www.intuit.ru';
alert("Счастливого пути");
}
else
{
alert("Ладно, оставайтесь");
defaultStatus='Что сделано, то сделано';
}}
</SCRIPT>
...и в команду <BODY>:
<BODY onLoad="go()">
```

Глава 5
ФУНКЦИИ И КОНЦЕПЦИЯ ОБЪЕКТОВ

В этой главе будут полностью рассмотрены функции и представлена концепция объектов в JavaScript.

В JavaScript функции также используются для многократного выполнения одной и той же задачи. До сих пор функции всегда вызывались вручную с помощью скобок: *myFunction()*. В JavaScript можно соединить функцию практически с любым событием, которое может порождать пользователь. Далее представлена функция, которая подсчитывает, сколько раз пользователь щелкнул на странице.

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

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

Результат представлен на рис. 5.1.

В первом примере *"clickCount++"*, единица добавляется к переменной *clickCount* после чтения ее значения. В случае *"++clickCount"* единица добавляется к переменной *clickCount* перед чтением ее значения. Так как в этом примере переменной *clickCount* в начале присваивается значение 0, то единицу к ней необходимо добавлять до задания значения поля ввода, поэтому использована запись *"++clickCount"*.

Предыдущий пример может показаться достаточно знакомым. Так же, как и раньше, определяется переменная и функция. Изменение состоит в том, что вместо вызова функции *documentClick()*

код содержит указание, что функция должна выполняться всякий раз, когда пользователь щелкает на документе. *"document.onclick "* связывает функцию с событием документа onclick (*"при щелчке"*).

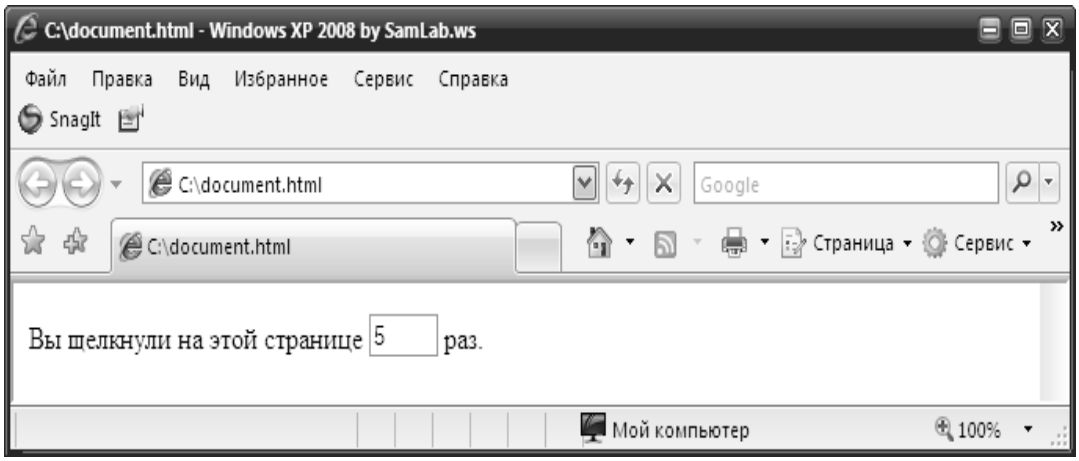

Рис. 5.1. Результат выполнения кода

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

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

Поскольку это поле ввода используется как счетчик, то нежелательно, чтобы пользователи щелкали на нем и изменяли его значение. Здесь на помощь приходит другое связывание, *"onfocus"*, которое срабатывает, когда курсор перемещается на объект.

Поэтому при щелчке на поле ввода или при перемещении на поле ввода с помощью клавиши Tab вызывается *"onfocus"*.

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

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

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

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

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

Результат выполнения кода представлен на рис. 5.2–5.3.

Вернемся к функциям и рассмотрим передачу функции аргументов. В предыдущем примере *"obj"* является аргументом. Пред-

полагается, что *"obj"* содержит указатель на поле ввода, на котором был произведен щелчок. В функцию можно передавать любое количество аргументов. Если потребуется 10 аргументов, то функция будет выглядеть следующим образом:

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

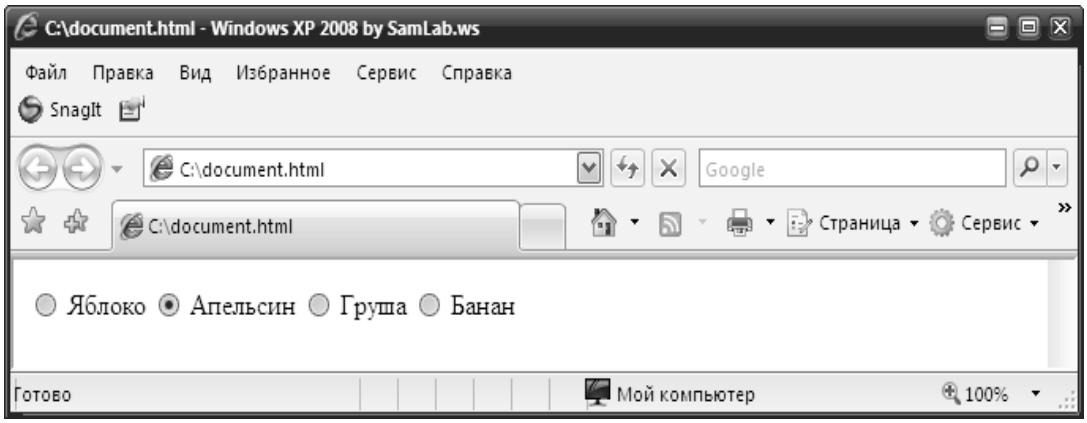

Рис. 5.2. Результат выполнения кода

Рис. 5.3. Результат выполнения кода

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

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

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

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

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

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

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

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

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

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

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

  return out;
}

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

  return out;
}

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

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

Результат представлен на рис. 5.4–5.5.

В этом небольшом фрагменте кода происходят следующие события. Вначале просто определяют две функции: multiply и add. Функция multiply просто перемножает все переданные ей числа.

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

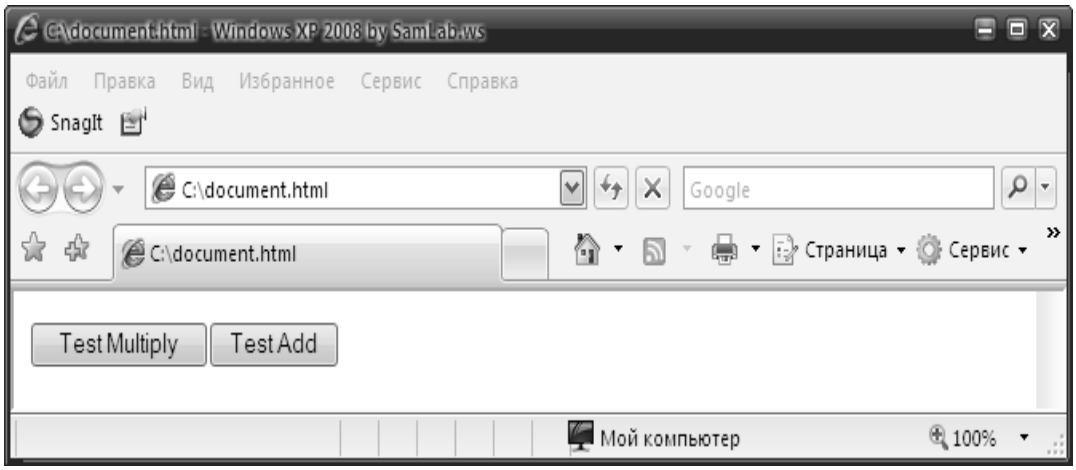

Рис. 5.4. Результат выполнения кода

Рис. 5.5. Результат выполнения кода

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

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

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

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

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

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

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

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

§ 5.1. Объектная модель документа

Этот параграф посвящен Объектной модели документа, или коротко DOM (Document Object Model). DOM является просто специальным термином для «всего на web-странице». Объектная модель

включает каждую таблицу, изображение, ссылку, поле формы и т. д. на web-странице. JavaScript позволяет манипулировать с любым элементом на странице в реальном времени. Можно скрывать или полностью удалять любой элемент, добавлять элементы, копировать их, изменять такие свойства, как цвет, ширина, высота, и т. д., а при некотором воображении можно даже реализовать функции перетаскивания, анимации и почти все остальное, что можно придумать.

Прежде всего необходимо понять, что с точки зрения браузера страница HTML является точно тем же, что и документ XML. Если читатель имеет опыт работы с XML, то сможет понять обработку DOM достаточно легко.

Изображение: блок-схема документа. Изображение может помочь лучше понять отношения предок – потомок в этом коде.

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

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

```
<img src="button_off.gif"
onmouseover="this.src='button_over.gif';"
onmousedown="this.src='button_down.gif';"
onmouseout ="this.src='button_off.gif';"
onmouseup  ="this.src='button_over.gif';">
```

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

§ 5.2. Добавление и удаление элементов

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

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

```
1  var inputs = 0;
2  function addContact(){
3    var table = document.getElementById('contacts');
4
5    var tr    = document.createElement('TR');
6    var td1   = document.createElement('TD');
7    var td2   = document.createElement('TD');
8    var td3   = document.createElement('TD');
9    var inp1  = document.createElement('INPUT');
10   var inp2  = document.createElement('INPUT');
11
12   if(inputs>0){
13     var img     = document.createElement('IMG');
14     img.setAttribute('src', 'delete.gif');
15     img.onclick = function(){
16       removeContact(tr);
17     }
18     td1.appendChild(img);
19   }
20
21   inp1.setAttribute("Name", "Name" +inputs);
22   inp2.setAttribute("Name", "Email"+inputs);
23
```

```
24   table.appendChild(tr);
25   tr.appendChild(td1);
26   tr.appendChild(td2);
27   tr.appendChild(td3);
28   td2.appendChild(inp1);
29   td3.appendChild(inp2);
39
31   inputs++;
32  }
33  function removeContact(tr){
34    tr.parentNode.removeChild(tr);
35  }
36  <table>
37    <tbody id="contacts">
38      <tr>
39      <td colspan="3"><a
href="javascript:addContact();">Добавьте контакт</a></td>
40      </tr>
41      <tr>
42        <td></td>
43        <td>Name </td>
44        <td>Email</td>
45      </tr>
46    </tbody>
47  </table>
```

Результат выполнения кода представлен на рис. 5.6.

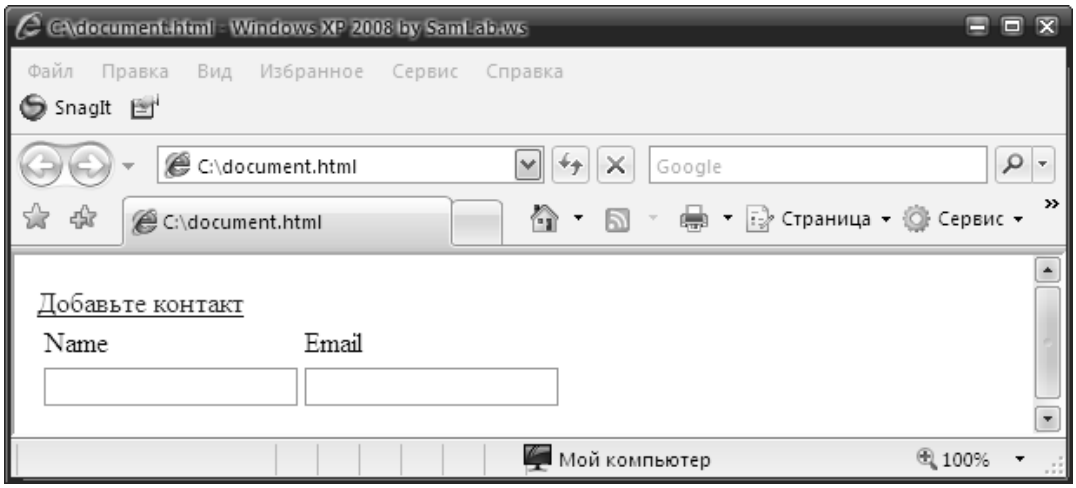

Рис. 5.6. Результат выполнения кода

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

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

```
<tr>
  <td>
    <img src="delete.gif">
  </td>
  <td>
    <input name="Name1">
  </td>
  <td>Если это
    <input name="Email1">
  </td>
</tr>
```

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

После создания всех этих элементов остается в действительности поместить их в документ. Каждый элемент на странице имеет встроенную функцию *"appendChild"*, которую можно использовать для добавления к этому элементу потомка. Когда добавляется потомок, то он добавляется как последний элемент, поэтому если таблица уже имеет в качестве потомков 10 тегов *TR* и добавляется еще один, то он будет добавлен как 11-й тег *TR*. Мы начинаем с получения ссылки на таблицу (строка 3). Затем мы

добавляем *TR* к этой таблице (строка 24) и добавляем затем 3 *TD* (строки 25–27). Второй и третий *TD* содержат поле ввода, поэтому мы добавляем эти поля ввода (28–29).

Теперь есть новый элемент *TR*, и он находится на странице. Осталось пояснить еще пару моментов. Чтобы форма была обработана правильно, все поля ввода должны иметь различные имена. Поэтому мы задаем имя двух полей ввода на основе счетчика (21–22), а затем увеличиваем счетчик (31). Это делается с помощью еще одной новой функции *setAttribute*, которая имеет два параметра: имя атрибута и значение атрибута. Для нее существует дополнительная функция *getAttribute*, которая имеет только один аргумент: имя атрибута, значение которого надо получить.

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

по сути то же самое, что

element.name="elementName"

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

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

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

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

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

§ 5.3. Элементы-потомки

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

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

```
function setColors(tbody, color1, color2){
  var colors  = [color1, color2];
  var counter = 0;
  var tr      = tbody.firstChild;
  while(tr){
    tr.style.backgroundColor = colors[counter++ % 2];
    tr = tr.nextSibling;
  }
}
Демонстрация
Row 1
Row 2
Row 3
Row 4
Row 5
Row 6
Color #1:  Color #2:  Раскрасьте таблицу
```

При рассмотрении этого небольшого фрагмента кода мало что нужно пояснять в том, как можно получить этот интересный небольшой эффект. Код начинается с получения ссылки на первый элемент TR в таблице с помощью метода *firstChild*. Затем каждый *TR* раскрашивается по очереди двумя разными цветами, используя *tr.style*. Цвет фона задается одним из двух цветов из массива *colors*. Если *counter* имеет четное значение, то цвет фона задается

как *color1*. Иначе он задается как *color2*. Это реализуется с помощью оператора деления по модулю (%). Для тех, кто забыл, напомним, что операция вычисляет остаток при делении. 5 / 2 = 2 с остатком 1. Поэтому 5 % 2 (5 по модулю 2) = 1.

Здесь достаточно сказать, что *element.style* предоставляет доступ ко всему, что можно задать с помощью таблицы стилей. Если нужно, например, задать стиль элемента, то можно прочитать/записать весь стиль с помощью *element.style.cssText*.

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

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

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

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

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

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

Обе эти функции делают то же самое, что и первая функция *setColors*, но написано это немного по-другому. Первая функция

использует атрибут *childNodes*. Как ранее говорилось, *childNodes* содержит массив, элементами которого являются потомки. Поэтому мы циклически перебираем *tbody.childNodes* и изменяем цвет каждого потомка, которые все должны быть элементами *TR*.

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

§ 5.4. Работа с текстом

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

* this is a link and an image: <img src="img.jpg"> *

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

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

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

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

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

Первая строка показывает, как создать текст с помощью *document.createTextNode*. Это аналогично функции использованной ранее функции *document.createElement*. Третья строка содержит еще одну новую функцию *insertBefore*, которая аналогична *appendChild*, за исключением того, что имеет два аргумента: добавляемый элемент и существующий элемент, перед которым надо сделать вставку. Так как мы хотим добавить новый текст перед элементом A и знаем, что элемент A является вторым элементом в *div*, то мы используем *ourDiv.childNodes[1]* в качестве второго аргумента для *insertBefore*.

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

В качестве последнего замечания к этому параграфу: одним из наиболее полезных средств при попытке протестировать или отладить код JavaScript, который изменяет DOM, является сценарий обхода дерева DOM. Проще говоря – это сценарий, который показывает каждый элемент и каждый атрибут объекта DOM.

Практические задания к главе 5

Задание 1

Перепишите приведенный выше код JavaScript в виде функции. Можно, при желании, изменить некоторые математические операции, например, на деление. Включите в тело документа HTML приветственное сообщение. Используйте для выполнения функции событие *onLoad*.

Возможное решение

```
<html>
<head>
```

```
<SCRIPT type="text/javascript">
function vars()
{
  numsums=10 + 2
  alert("10 + 2 is " + numsums)
  var x = 10
  alert("ten is " + x)
  y = x * 2
 alert("10 X 2 = " + y)
  z = "Привет " + "Пока"
 alert(z)
}
</SCRIPT>
</head>
<BODY OnLoad="vars()">
<h1>Добро пожаловать на мою страницу</h2>
</body>
</html>
```

Мы будем заниматься переменными очень много по мере изучения новых команд Java Script. Главное, пытайтесь понять, что делаете, а не копируйте автоматически.

Задание 2

Создайте страницу HTML. Разместите все по центру страницы. Используйте тег H1 со своим именем. Под ним поместите изображение Bubble1.gif. Когда курсор мыши укажет на это изображение, оно должно измениться на изображение Bubble2.gif. Когда курсор мыши сместится с этой ссылки, снова должно появиться изображение Bubble1.gif.

Возможное решение

Например, следующий код:

```
<html>
<head>
</head>
<body bgcolor="white">
<center>
```

```
<h1>Иван Иванович</h1>
<p>
<a href="" onMouseOver="document.pic1.src='bubble2.gif' "
   onMouseOut="document.pic1.src='bubble1.gif'">
<img src="bubble1.gif" border=0 name="pic1"></a></p>
</body>
</html>
```

Существует множество разных трюков с *onMouseOver* и *onMouseOut*.

Глава 6
ПРИМЕРЫ СЦЕНАРИЕВ

В этой главе рассмотрены примеры решения некоторых задач. Не все из них одинаково часто возникают при разработке приложений и, в частности, веб-страниц, однако их изучение позволит получить практические навыки программирования.

§ 6.1. Простые визуальные эффекты

6.1.1. Смена изображений

В веб-дизайне часто возникает необходимость заменить одно изображение на другое. Новички нередко начинают свое творчество именно с этой операции. Поэтому рассмотрим данную задачу подробнее.

Суть смены изображений заключается в том, чтобы с помощью сценария изменить значение атрибута *SRC* тега *<IMG>*. Напомним, что атрибут *SRC* имеет в качестве значения строку, указвающую месторасположение графического файла. Если элемент *<IMG>*, задающий изображение, содержится в HTML-документе, то в объектной модели имеется объект этого элемента со свойством src. Значение этого свойства можно изменить в сценарии. При этом в окно браузера загружается соответствующий графический файл, если, разумеется, он будет найден. В следующем примере щелчок на изображении из файла pict1.gif заменяет его изображением из файла pict2.gif. Поскольку сценарий очень небольшой, он записан в строке, которая присваивается атрибуту onclick тега *<IMG>*.

```
<HTML>
<IMG ID = "myimg" SRC = 'pict1.gif'
Onclick = "document.all.myimg.src = 'pict2.gif'>
</HTML>
```

В приведенном выше примере смена изображения происходит лишь при первом щелчке на нем. Последующие щелчки не приведут к видимым изменениям, поскольку второе изображение будет заменяться им же. Чтобы повторный щелчок приводил к отображению предыдущего рисунка, сценарий необходимо слегка усложнить: следует создать переменную-триггер (так называемый флаг), принимающий одно из двух возможных значений. По текущему значению флага сценарий может определить, какое именно из двух изображений следует отобразить. После смены изображения необходимо изменить и значение флага. Далее приведен вариант кода:

```
<HTML>
<IMG ID = "myimg" SRC = 'pict1.gif onclick = "imgchange()">
<SCRIPT>
var flag=false                  // флаг (триггер)
function imgchange() {          // обработчик щелчка на изображении
if (flag) document.all.myimg.src = "pict1.gif"
else document.all.myimg.src = "pict2.gif"
flag=!flag //изменяем значение флага на противоположное
}
</SCRIPT>
</HTML>
```

Далее создадим одну функцию-обработчик, которая кроме всего прочего будет сама определять, на каком именно изображении произошел щелчок. Идентификаторы ID тегов *<IMG>* зададим некоторым регулярным образом (например *"10"*, *"11"*, *"12"*, ...). Так часто поступают при использовании массивов. Создадим еще два массива, содержащих имена графических файлов: один для исходных изображений, а другой – для замещающих. Наконец, HTML-документ с изображениями сгенерируем с помощью сценария. Для этого сначала сформируем строку, содержащую теги *<IMG ...>*, а затем запишем ее в документ.

Код программы для увеличения миниатюр при щелчке мыши представлен ниже:

```
<HTM>
<SCRIPT>
var apictl = new Array("pict1.gif" ,...) /* массив имен исходных
файлов */
```

```
var apict2 = new Array("pict2.gif" ,...) /* массив имен замещающих файлов */
var aflag = new Array(apict1.length) // массив флагов
/* формирование строки тегов, описывающих изображения */
var xstr = " "
for(i=0; i<apict1.length;i++){
xstr+='<IMG ID = "i'+i+'" SRC = "'+apictl[i]+'"onclick="imgchange()">'
}
document.write(xstr) // запись в документ
function imgchange() { // обработчик щелчка на изображении
var xid = event.srcElement.id /* id изображения, на котором был щелчок */
var n = parselnt(xid.substr(1)) //выделяем номер элемента
if (aflagtn])
document.all[xid].src = apict1[n]
else
document.all[xid].src = apict2[n]
aflag[n] = !aflag[n] /* изменяем значение флага на противоположное */
|
</SCRIPT>
</HTML>
```

Обратите внимание, как происходит обращение к свойству *src*: *document.allfxid].src*, а не *nt.all.xid.src* и тем более не *document.all["xid"].src.* Это объясняется тем, что *xid* является строковой переменной, содержащей значение идентификатора ID, а не собственно значением идентификатора.

6.1.2. Подсветка кнопок и текста

Рассмотрим задачу изменения цвета кнопки при наведении на нее указателя мыши. При удалении указателя с кнопки должен вернуться ее первоначальный цвет. Это так называемая подсветка кнопок. Сразу займемся общим случаем нескольких кнопок. В приводимом примере имеется три элемента, задающие кнопки, которые находятся в контейнере формы *<FORM>*. К этому контейнеру привязываются обработчики событий onmouseover (наведение указателя мыши) и onmouseout (удаление указателя мыши).

Таким образом, инициатором (получателем) этих событий может быть любой элемент формы (в нашем примере – любая из трех кнопок). В обычном состоянии кнопки имеют серый цвет, заданный 16-м кодом aOaOaO. При наведении указателя мыши цвет кнопки становится желтым (yellow).

```
<HTML>
<STYLE>
mystyle {font-weight: bold ; background-color: aOaOaO}
</STYLE>
#
 <FORM onmouseover = "colorchange('yellow')" onmouseout =
"colorchange('aOaOaO')">
<INPUT TYPE = "BUTTON" VALUE = "Первая" CLASS =
"mystyle" onclick =
"alert ('Вы нажали кнопку') ">
<INPUT TYPE = "BUTTON" VALUE = "Вторая" CLASS =
"mystyle" onclick =
"alert ('Вы нажали кнопку 2')">
<INPUT TYPE = "BUTTON" VALUE = "Третья" CLASS =
"mystyle" onclick =
"alert('Вы нажали кнопку 3')">
</FORM>
<SCRIPT>
function colorchahge(color) {              // изменение цвета кнопок
if(event.srcElement.type == "button")
event.srcElement.style.backgroundColor = color;
|
</SCRIPT>
</HTML>
```

Здесь в функции *colorchange()* проверяется, является ли инициатор события объектом типа *button* (кнопка). Если это так, то цвет кнопки изменяется, в противном случае – нет. Без этой проверки изменялся бы цвет не только кнопок, но и фона.

Аналогичным образом можно изменять цвет и других элементов, например фрагментов текста. Если требуется подсвечивать текст, то он должен быть заключен в какой-нибудь контейнер, например в теги <*P*>, <*B*>, <*I*> или <*DIV*>.

6.1.3. Мигающая рамка

Можно создать прямоугольную рамку, окаймляющую некий текст, которая периодически изменяет цвет. Иногда этот эффект используют для привлечения внимания пользователей. Рамка создается тегами одноячеечной таблицы с заданием нужных атрибутов и параметров стиля. Далее необходимо создать функцию, изменяющую цвет рамки таблицы на другой, и передать ее в качестве первого параметра методу *setInterval()*. Второй параметр этого метода задает период в миллисекундах, с которым вызывается функция, указанная в первом параметре.

В приведенном ниже примере рамка изменяет цвет с желтого на красный с периодом 0,5 с.

```
<HTML>
<TABLE ID = "mytab BORDER=1 WIDTH=150 style = "border : 10 solid:yellow">
<TR><TD> Мигающая рамка </TD></TR>
<TABLE>
<SCRIPT>
function flash() {          // изменение цвета рамки,
if ((document.all)          // если в документе ничего нет
return null;
if (mytab.style.borderColor == 'yellow')
mytab.style.borderColor = 'red'
else
mytab.style.borderColor = 'yellow';
setInterval("flash() " , 500); // мигание рамки с интервалом 500 мс
</SCRIPT>
</HTML>
```

6.1.4. Ссылки

Привлечь внимание посетителей к ссылкам на веб-странице можно с помощью эффекта динамического изменения их цвета. Мы не будем обсуждать здесь целесообразность этого метода, а сосредоточимся лишь на технической стороне. Суть задачи состоит в том, чтобы случайным образом выбирать и устанавливать цвет ссылок. В листинге 6.1 множества цветов, из которых происходит выбор, различаются для уже использованных и еще не использованных ссылок. Эти множества цветов задаются в виде массивов.

Листинг 6.1. Код для динамического изменения цвета ссылок

```
<HTML>
<A HREF="http://www.admiral.ru/~dunaev/i_is.Шm">Начало</A>
<A HREF="http://www.admiral.ru/~dunaev/examples.1nm_m">
Примеры HTML и JavaScript</A>
<A HREF="http://www.admiral.ru/~dunaev/mybook.htm">Мои
книги</A>
<SCRIPT>
Aclrlink = new Array()          /* массив цветов неиспользованных ссылок */
aclrlink[0]='yellow'
aclrlink[1]='#80FF80'
aclrlink[2]='#FFFF80'
aclrlink[3]='#408000'
aclrvlink k= new Array()        /* массив цветов использованных ссылок */
aclrvlink [0] ='blue'
aclrvlink [1) ='purple'
aclrvlink [2] ='black'
aclrvlink [3] ='red'
function colorchange() { // изменение цвета
link = Math.round((aclrlink.length+0.l)*Math.random())
vlink = Math.round ((aclrvlink.length+0.l)*Math.random())

document.linkColor = aclrlink[link]
document.vlinkColor = aclrvlink[vlink]
}
setInterval ("colorchange() ", 500)     // изменение цвета через 500 мс
</SCRIPT>
</HTML>
```

Заметим, что в результате вставки этого сценария в HTML-документ все ссылки документа начинают переливаться разными цветами, поскольку мы изменяем свойства *InkColor* и *vlinkColor* объекта *document*.

6.1.5. Объемные заголовки

Идея создания объемного заголовка довольно проста: достаточно несколько надписей с одинаковыми содержанием наложить

друг на друга с некоторым сдвигом по координатам. Потребуются как минимум две такие надписи. Одна из них предназначена для создания эффекта тени (задний план), а вторая располагается над первой. С технической точки зрения нужный эффект получается применением таблиц стилей. В листинге 6.2 приведен пример HTML-документа, в котором объемный заголовок создается с помощью трех наложенных друг на друга надписей. Таблица стилей определяется для тега абзаца *<P>*. В данной таблице задаются цвет и параметры шрифта надписей. Позиционирование надписей производится параметрами атрибута *STYLE* контейнерных тегов *<DIV>*, в которые заключены теги абзаца.

Листинг 6.2. Код для создания объемного заголовка

```
<HTML>
<HEAD>
<TITLE> 3d эффект </TITLE> <HEAD>
<! Каскадная таблица стилей для абзаца>
<STYLE>
P {font-family:sans-serif;font-size:72;font-weight:800; color:00aaaa}
P.highlight {color: siIver}
P.shadow {coloridarkred}
</STYLE>
<BODY BGCOLOR = aeb98c>
<! Тень>
<DIV STYLE = "position: absolute; top:5; left:5">
<P CLASS = shadow> Объемный заголовок</P>
</DIV>
<\ Подсветка>
<DIV STYLE = "position: absolute; top:0;left:0" >
<P CLASS = highlight > Объемный заголовок</P>
</DIV>
<! Передний план>
<DIV STYLE = "position:absolute; top:2; left:2">
<P>Объемный заголовок</P>
</DIV>
</BODY>
</HTML>
```

6.1.6. Применение фильтров

С помощью так называемых фильтров каскадных таблиц стилей можно получить разнообразные интересные визуальные эффекты. Например, постепенное появление (исчезновение) рисунка, плавное преобразование одного изображения в другое, задание степени прозрачности и т. п. В большинстве случаев веб-дизайнеры добиваются подобных эффектов с помощью обработки изображений средствами графических редакторов, таких как Adobe Photoshop, Macromedia Flash и др. При этом графика для веб обычно сохраняется в файлах формата gif, png, jpeg и swf.

Формат GIF очень популярен в веб-дизайне. Поддерживает чересстрочную загрузку, прозрачность пикселов и анимацию. Однако глубина цвета в GIF-файлах ограничена лишь 256 цветами, а пикселы могут быть либо полностью прозрачными, либо полностью непрозрачными (полупрозрачности быть не может).

Формат PNG имеет много общего с форматом GIF, но позволяет сохранять полноцветные изображения и не поддерживает анимацию.

Формат JPEG – еще один графический формат, часто используемый в веб-дизайне. Позволяет сохранять полноцветные изображения фотографического качества и загружать их в чересстрочном режиме, но не поддерживает прозрачность и анимацию.

В формате SWF сохраняется векторная графика и анимация, созданные в пакете Macromedia Flash, а также импортированные растровые изображения и звуковое сопровождение. Прозрачность изображений здесь может принимать множество значений от 0 до 100.

Во многих случаях при создании небольших анимаций (например, баннеров) и других не слишком сложных визуальных эффектов можно вполне обойтись средствами таблиц стилей и JavaScript, достигая своих целей при существенно меньших затратах ресурсов (объем файлов, время на разработку). При разработке более сложных проектов фильтры можно комбинировать с другими средствами.

Фильтр следует понимать как некий инструмент преобразования изображения, взятого из графического файла и вставленного в HTML-документ с помощью тега <IMG>. Однако следует иметь в виду, что фильтры работают только в IE4+.

Прозрачность. Прозрачность графического объекта в диапазоне целочисленных значений от 0 до 100 можно установить с помощью

фильтра alpha. Значение 0 соответствует полной прозрачности изображения, то есть оно становится невидимым. Значение 100 соответствует, наоборот, полной непрозрачности изображения. Поэтому правильнее говорить, что alpha является степенью непрозрачности. Сквозь прозрачные графические объекты видны нижележащие изображения. Степень их видимости определяется значением прозрачности изображения, лежащего выше. Кроме того, прозрачность имеет несколько вариантов градиентной формы. Например, можно сделать так, чтобы прозрачность постепенно увеличивалась от центра к краям изображения. Это позволяет просто и весьма эффективно согласовать вставляемое в документ изображение с фоном. Веб-дизайнеры хорошо знают, что выполнение такой операции с помощью графического редактора требует определенных навыков и усилий.

Фильтр *alpha*, как и другие, задается с помощью каскадной таблицы стилей и имеет ряд параметров. В следующем примере для графического изображения стиль определяется с помощью атрибута *STYLE*

```
<IMG ID = "myimg" SRC = "pict.gif"
STYLE = "position:absolute; top:10; left:50;
filter: alpha (opacity = 70. style = 3)">
```

Здесь параметр opacity определяет степень прозрачности (точнее, непрозрачности) как целое число в диапазоне от 0 до 100. Параметр *style* задает градиентную форму распределения прозрачности по изображению как целое число от 0 до 3. Если параметр *style* не указан или имеет значение 0, то градиент не применяется. Фильтр alpha имеет и другие параметры, определяющие прямоугольную область изображения, к которой применяется фильтр. По умолчанию он применяется ко всему изображению.

Фильтр можно определить в каскадной таблице стилей внутри контейнерного тега <*STYLE*> с помощью ссылки, имеющей следующую структуру:

```
#1c) изображения {filter: имя _фильтра (параметры)}
```

Доступ к свойствам фильтра в сценарии определяется следующим образом:

```
t.all.1c1_изображения.filters ["имя_фильтра"].параметр = значение
```

Для рассматриваемого случая это выражение может иметь, например, такой вид:

document.all.myimg.filters ["alpha"].opacity = 30

Для IE5.5+ можно использовать другой синтаксис, рекомендованный Microsoft. Его особенность в том, что в каскадной таблице стилей задается ссылка на специальный компонент и имя фильтра:

#1c1_изображения {filter:progid:DXImageTransform.Microsoft.имя_фильтра (параметры)}

Пример

#myimg {filter: progid: DXImageTransform.Microsoft.alpha (opacity = 70, style = 3)}

Доступ к свойствам фильтра в сценарии в новом синтаксисе определяется следующим образом:

document.all.1й_изображения.filters["DXImageTransform.Microsoft.имя_фильтра"].параметр = значение

Пример

document.all.myimg.filters["DXImageTransform.Microsoft.alpha"].opacity = 30

Трансформация. Фильтры преобразования изображений позволяют организовать постепенное появление (исчезновение) изображения, а также трансформацию одного графического объекта в другой. Это так называемые динамические фильтры. В отличие от статических фильтров (например, *alpha*), их применение обязательно связано со сценариями. Суть преобразования графического объекта заключается в том, что сначала необходимо зафиксировать первое изображение, затем выполнить замену этого изображения другим и/или изменить параметры того же самого изображения, а после этого выполнить собственно трансформацию. Все эти действия выполняются в сценарии. Фиксация и трансформация изображения производятся с помощью специальных методов фильтра – соответственно *apply()* и *play()*. При желании остановить процесс преобразования можно с помощью метода *stop()*.

Для преобразования изображений используется фильтр *revealtrans*. Он имеет следующие параметры:

• **duration** – длительность преобразования в секундах (число с плавающей точкой);

• **transition –** тип преобразования (целое число от 0 до 23); типу 23 соответствует один из типов от 0 до 22, выбранный случайным образом.

Сначала рассмотрим применение фильтра revealtrans для создания эффекта появления изображения. Ниже приводится пример, в котором изображение постепенно появляется после загрузки документа, то есть по событию *onload*:

```
<HTML>
<BODY onload = "transform()">
<IMG ID = "myimg" SRC = "pict.gif" STYLE = "position:absolute; top:10; left :5 0; visibility = "hidden" filter:revealtrans(duration= 3 , transition = 12)">
</BODY>
<SCRIPT>
function transforraO{                    // появление изображения
document.all.myimg.style.visibility = "hidden" /* делаем изображение невидимым */
myimg. filters C'revealtrans").apply() /* фиксируем исходное состояние изображения */
myimg.style.visibility = "visible"       /* делаем изображение видимым */
myimg.filtersC 'revealtrans') .play()    /* выполняем преобразование */
</SCRIPT>
</HTML>
```

В этом примере, в функции *transform()*, делаем изображение, заданное в HTML-документе тегом *<IMG>*, сначала невидимым, поскольку хотим, чтобы оно возникало из небытия, а не исчезало. Затем с помощью метода *apply()* мы фиксируем его как исходный кадр в цепочке кадров преобразования. Далее готовим конечный кадр – просто делаем изображение видимым. Наконец, применяем фильтр с помощью метода *play()*. Использовался тип преобразования *(transition = 12)*, соответствующий эффекту плавной трансформации.

Более в общем случае: трансформация одного графического объекта в другой. В отличие от рассмотренного выше случая, эта задача сводится к установке начального и конечного изображений. Это делается путем присвоения нужных значений свойству src объекта, соответствующего изображению.

```
<HTML>
<BODY onload = "transform2() ">
<IMG ID = "myimg" SRC = "pictl.gif" STYLE = "position : absolute ; top:10; left:50; fiIter: revealtrans(duration = 3, transition = 12)
</BODY>
<SCRIPT>
function transform2(){ myimg.filters ("revealtrans").apply И
"pict2.gif" myimg.filters("revealtrans").play ()
</SCRIPT>
</HTML>
```

В следующем примере зададим трансформацию изображения, которая происходит при щелчке на графическом объекте. При первом щелчке на изображении файла pict1.gif трансформируется в рисунок из файла pict2.gif. При следующем щелчке, наоборот, второй рисунок трансформируется в первый и т. д.

```
<HTML>
<IMG ID = "myimg" onclick = " transforms()" SRC = "pict1.gif"
STYLE = "position:absolute; top:10; left: 50; filter: revealtrans (duration = 3,transition = 12)">
<SCRIPT>
function transoformB(){
                                    // преобразование
                                    // фиксируем исходное изображение
if (document.all.myimg.src.indexOf("pict1") ! = -1) /* определяем конечное изображение */
document.all.myimg.src = "pict2.gif"
else
document.all.myimg.src="pict1.gif"myimg.filters("revealtrans").play()
                                    // выполняем преобразование
}
</SCRIPT>
</HTML>
```

Свойство src содержит полный URL-адрес графического объекта, даже если мы присваиваем ему только имя файла. Поэтому в теле функции *transform3()* приходится выяснять, входит ли строка с именем файла в URL-адрес, а не проверять, равен ли URL-адрес имени файла.

Рассмотренный выше синтаксис выражений воспринимается браузерами IE4 +. Для IE5.5+ можно использовать другой синтаксис, рекомендованный Microsoft.

Повороты. В браузерах IE5.5+ возможно применение фильтра basicimage, имеющего множество параметров, с помощью которых изображение можно повернуть на угол, кратный 90°, задать прозрачность, зеркально отразить, определить маску и др. В IE4+ все эти эффекты, за исключением поворотов, создаются отдельными фильтрами. Порядок их применения такой же, как и фильтра alpha, задающего прозрачность. Здесь мы рассмотрим создание эффекта поворота.

Параметр rotation фильтра basicimage принимает целочисленные значения: 0 (нет поворота), 1 (90°), 2 (180°), 3 (270°). В примере, приведенном ниже, в результате щелчка на изображении оно поворачивается на угол 90°.

```
< html >
<STYLE>
#myimg {filter: progid:DXImageTransform. Microsoft. basicimage}
</STYLE>
<IMG ID = "myimg" src = "pict.gif" onclick = "rotor()">
<SCRIPT>
function rotor(){      // повороты
var r =
document. all.myimg.fiIters["DXImageTransform.Microsoft. basicimage"] .rotation if (r == 3)
r = 0
else
r++
document.all.myimg.fiIters ["DXImageTransform.Microsoft.basicimage"].rotation = r
</SCRIPT>
```

Применение нескольких фильтров одновременно. К видимому элементу можно применить несколько различных фильтров одновременно. В следующем примере мы делаем графический

объект полупрозрачным с помощью фильтра alpha и трансформируемым с помощью фильтра *revealtrans*. Полупрозрачным изображение становится сразу при загрузке в браузер, а трансформация происходит при щелчке на нем. Вот вариант кода, реализующий данный сценарий:

```
<HTML>
<IMG ID = "myimg" onclick = "transforms i " SRC = "earth.gif"
STYLE = "position.-absolute; top:10; left:50;
uration = 3, transition = 12), alpha(opaci ty = 50)"
<SCRIPT>
function transform3() {    // преобразование изображения
if (document. all .myimg.src . indexOf ( "pictl") !=-1)
/* определяем конечное изображение */
document. all.myimg.src = "pict2.gif"
else
document.all.myimg.src="pictl.gif" myimg.fliters("revealtrans").play()
/* выполняем преобразование */
|
</SCRIPT>
</HTML>
```

6.1.7. Эффект печати на пишущей машинке

Постепенный вывод на страницу текста (эффект печати на пишущей машинке) можно создать на основе использования метода *setInterval()*. Об этом и других методах управления во времени уже говорилось в предыдущих параграфах.

Ниже приводится HTML-документ с заголовком и текстовой областью, задаваемой тегом <*TEXTAREA*>. Сценарий в этом документе выводит в текстовую область символы некоторой строки с задержкой 0,1 с. После завершения вывода всей строки этот процесс останавливается. Функция *wrtext()* просто формирует строку, которую требуется вывести в текстовой области в данный момент. Собственно вывод производится путем присвоения значения свойству *value* текстовой области. Функция *wrtext()* передается в виде строкового параметра методу *setInterval()*, который и вызывает ее периодически с интервалом, указанным в миллисекундах, в качестве второго параметра. В примере этот интервал равен 100. Метод *setInterval()* возвращает целочисленный идентификатор запущенного процесса, который мы сохраняем в переменной *xinterval*.

Это значение передается методу *clearInterval()* как параметр, чтобы завершить процесс периодического вызова функции *wrtext()*, когда «напечатается» вся строка.

```
<HTML>
<H1>Моя веб-страница</H1>
<TEXTAREA ID = "mytext" ROWS = 8 COLS = 25></TEXTAREA>
<SCRIPT>
var mystr = "Привет, мои друзья! Рад вам сообщить приятное известие"
var astr = mystr.split ("")   // разбиваем строку на массив
// символов var typestr = "" var i = 0
var xinterval = setInterval ("wrtext ( ) " , 100) /* периодический вызов функции wrtext() */
function wrtext(){ // вызывается с помощью метода setlnterval ()
if (i < astr.length) {
typestr+=astr[i]    // выводимая строка
document.all.mytext.value = typestr // вывод строки i+ +
}else
clearInterval(xinterval)  // прекращаем
} </SCRIPT>
```

Другие примеры использования метода *setInterval()* вы найдете в следующем параграфе.

§ 6.2. Движение элементов

Перемещение видимых элементов HTML-документа в окне браузера основано на изменении значений параметров позиционирования top и left таблицы стилей. Можно указать эти параметры в атрибуте *STYLE* или в теге <*STYLE*> для тех тегов, которые задают видимые элементы (изображения, тексты, ссылки, кнопки и т. п.). Затем остается только определить в сценарии способ изменения параметров координат top и left для того или иного элемента. Можно заставить элемент перемещаться постоянно, в течение заданного времени или в ответ на события (например, по щелчку кнопкой мыши, при наведении указателя мыши на элемент и т. п.).

Оператор цикла для организации расчета координат обычно не применяется, поскольку, пока выполняется цикл, другие выражения сценария не работают.

Чтобы распараллелить вычислительные процессы, используют методы *setInterval()* и *setTimer()*.

6.2.1. Движение по заданной траектории

Схема сценария, осуществляющего непрерывное перемещение видимого элемента документа, имеет следующий вид:

```
function init_move() {  // инициализация движения
...     // подготовка к запуску функции move()
setInterval("move()", задержка)
function move(){
...      /* изменение координат top и left стиля перемещаемого элемента */
// вызов функции для перемещения элемента
```

Таким образом, создаются две функции, имена которых могут быть произвольными. Первая функция, *init_move()*, осуществляет подготовку исходных данных и вызывает метод *setInterval()* с указанием в качестве первого параметра имени второй функции *move()* в кавычках. Вообще говоря, первый параметр метода *setInterval()* является строкой, содержащей выражение, которое должно выполняться периодически во времени. Вторая функция, *move()*, изменяет координаты элемента. Поскольку метод *setInterval()* вызывает функцию *move()* периодически через заданное количество миллисекунд, то координаты элемента изменяются постоянно. При этом создается эффект движения. Скорость и плавность движения зависят от величин приращения координат (в функции *move()*) и временной задержки (второго параметра метода *setInterval()*). Чтобы начать перемещение элемента, необходимо просто вызвать первую функцию, *init_move()*.

Линейное движение. Рассмотрим в качестве примера сценарий линейного перемещения изображения, то есть движения по прямой линии.

```
<HTML>
<IMG ID="myimg" SRC="pict.gif" STYLE="position:absolute; top:10;left:20">
```

```
<SCRIPT>
function init_move() {
dx = 8     // приращение по y
dy = 3     // приращение по x
// периодический вызов функции move ()
function move() {    // изменение координат изображения
/* Текущие координаты: */
var y = parseInt(document.all.myimg.style.top)
var x = parseInt(document.all.myimg.style.left)
/* Новые координаты: */
document.all.myimg.style.top = y+dy
document.all.myimg.style.left = x+dx
}
init_move()     // начинаем движение
</SCRIPT>
</HTML>
```

Переменные *dx* и *dy* являются глобальными и поэтому видны в функции *move()*. Если бы мы определили их в *init_move()* с помощью ключевого слова *var*, то они стали бы локальными и недоступными в *move()*. Заметьте также, что значения параметров позиционирования в таблице стилей имеют строковый тип. Поэтому нам потребовалось применение функции *parseInt()* для приведения их к числовому типу.

Остановка движения. Рассмотренные выше сценарии обеспечивают непрерывное перемещение элемента документа. Поскольку это перемещение осуществляется по прямой линии, то рано или поздно элемент выйдет за пределы окна браузера. Разумеется, можно усовершенствовать функцию *move()* таким образом, чтобы она удерживала элемент в некоторой области окна браузера, изменяя в случае необходимости, например, знак приращений координат. Однако может потребоваться, чтобы элемент пересек окно браузера и больше не возвращался. Для остановки движения уже невидимого элемента служит метод *clearInterval()*, единственным параметром которого является целочисленный идентификатор, возвращаемый методом *setInterval()*.

Чтобы иметь возможность прекратить движение элемента, следует сохранить значение, возвращаемое методом *setInterval()*, в глобальной переменной, а затем использовать его в качестве

параметра метода *clearInterval()* в теле функции *move()*. Напомним, что *move()* – функция, имя которой передается в качестве первого параметра методу *setInterval()*. Сценарий с запуском и остановкой движения может выглядеть, например, следующим образом:

```
function init_move(xid, dx, dy) {
var prmstr = ""+xid+" ,"+dx+" ,"+dy  /* строка параметров для move() */
prmstr = "move(" +prmstr+ ")"
idjnove = setInterval (prmstr , 200)   /* сохраняем идентификатор движения */
}
function move(xid, dx , dy) {
y = parseInt(document.all[xid].style.top)
x = parseInt(document.all[xid].style.left)
document.all.myimg.style.top = y+dy
document.all.myimg.style.left = x+dx
if (parseInt(document.all[xid].style.left) > 350) /* остановка по условию */
}
clearInterval(id move)
init_move("myimg", 10, 5)    // начинаем движение
```

В этом примере движение остановится, как только горизонтальная координата элемента превысит 350 пикселов.

Движение по эллипсу. Для организации движения по замкнутой траектории, в качестве примера возьмем эллипс, поскольку его легко модифицировать. Движение по эллипсу задается несколькими параметрами, такими как большая и малая полуоси, положение центра и угол поворота относительно горизонтали, угловая скорость перемещения и др. В частном случае, когда одна из полуосей эллипса равна нулю, траектория вырождается в прямую линию. При этом движение будет происходить то в одну, то в другую сторону. Очевидно, в случае равенства полуосей траектория приобретает вид окружности.

В листинге 6.3 представлены определения двух функций, задающих движение по эллипсу. Функция *ellipse()* является функцией инициализации движения, а *move()* – функция, передаваемая методу *setInterval()*.

Листинг 6.3. Определения двух функций, задающих движение по эллипсу

```
function ellipse(xid, alpha, a, b, omega, x0, y0, ztime.dt) {
/* Движение по эллипсу
Параметры:
xid – id движущегося объекта, строка
alpha – угол поворота эллипса, град.
x0 – x-координата центра эллипса, пикселы
y0 – y-координата центра эллипса, пикселы
a – большая полуось эллипса, пикселы
b – малая полуось эллипса, пикселы
omega – угловая скорость, град/с; знак задает направление вращения
ztime – начальная фаза, град.
dt – временная задержка, с */
проверка наличия параметров:
if(!alpha) alpha = 0
if(!a) a=0
if(!b) b=0
if(!omega) omega = 0
if(!x0) x0 = 0
if(!y0) y0 = 0
if(!ztime) ztime = 0
if(!dt) dt = 0
var t = –ztime        /* чтобы начальное значение было 0, поскольку в move() уже есть приращение */
setlnterval("move(" + xid + " ," + alpha + " ," + a +" ," + b + " ," + omega + " ," + x0 + " ," + y0 + " ," + ztime + " ," + dt + " ) " , ztime*1000) /* многократный вызов move() с интервалом ztime, мс */
{
function move(xid, alpha, a, b, omega, x0, y0, ztime, dt)
/* пересчет координат,
вызывается из ellipse()*/
t+= ztime
/* x,y – координаты в собственной системе координат */
var x = a*Math.cos((omega*t + dt)*Math.PI/180)
var y = b*Math.sin((omega*t + dt)*Math.PI/180)
var as = Math.sin(alpha*Math.PI/180)
var ac = Math.cos(alpha*Math.PI/180)
document.all [xid].style.top = -x*as + y*ac + y0
document.all [xid].style.left = x*ac + y*as + x0
```

Движение по произвольной кривой. Рассмотрим задачу организации движения видимого элемента по произвольной кривой, заданной выражением с одной переменной, указав функции движения в качестве параметров не одно, а два выражения, которые описывают изменения вертикальной и горизонтальной координат элемента. Эти выражения будут содержать одну переменную, которую мы обозначим через *x* – строчной латинской буквой. Переменную *x* можно интерпретировать как независимый параметр движения (например, время). С помощью встроенной функции *eval()* можно вычислить значения этих выражений при конкретном значении переменной *x* и присвоить их параметрам left и top таблицы стилей перемещаемого элемента. Функция (пусть это будет *move()*), которая все это выполняет, передается в качестве первого параметра методу *setInterval()*, который периодически вызывает ее через заданный интервал времени.

Итак, функция инициализации движения *curvemove()* будет принимать три строковых и один числовой параметр. Строковые параметры содержат соответственно значение идентификатора ID перемещаемого элемента, выражение для вертикальной координаты и выражение для горизонтальной координаты. Числовой параметр определяет период времени, через который координаты элемента пересчитываются. Ниже приводятся определения функций *curvemove()* и *move()*:

```
function curvemove(xid, yexpr, xexpr, ztime) {
/* Движение по произвольной кривой. Версия 0.
Параметры:
xid – id движущегося объекта, строка
yexpr – выражение для вертикальной координаты
xexpr – выражение для горизонтальной координаты
ztime – интервал времени между вызовами функции move(), мс */
if (!xid) return null
if (!yexpr) yexpr = "x"
if (!xexpr) xexpr = "x"
if (!ztime) ztime = 100  // интервал времени, мс
x = 0                          /* глобальная переменная, входящая в выражения yexpr и xexpr */
setInterval("move('"+xid+" ', + yexpr + " "+ xexpr +"' ztime)
}
```

```
function move(xid, yexpr, xexpr) {
x++
document.all[xid].style.top = eval(yexpr)
document.all[xid].style.left = eval(xexpr)
}
```

Переменная *x* в функции *curvemove()* является глобальной и поэтому доступна в функции *move()*. Чтобы сделать переменную *x* глобальной, мы просто не использовали ключевое слово *var* перед первым ее появлением в коде.

Исходное позиционирование перемещаемого элемента с помощью таблицы стилей не играет особой роли, поскольку при вызове функции *curvemove()* элемент помещается в точку с координатами, равными значениям выражений xexpr и yexpr, вычисленным при *x* = 0. Поэтому начальное позиционирование следует задавать с помощью соответствующего выбора этих выражений. Ниже приведен пример HTML-документа с движущимся изображением.

```
<HTML>
<IMG ID = "myimg" SRC = "pict1.gif" STYLE = "position:absolute">
<SCRIPT>
function curvemove(xid, yexpr, xexpr, ztime) {
// код определения функции
}
function move(xid, yexpr, xexpr) {          // код определения функции
}
Curvemove()'myimg' , "100 + 50*Math.sin(0.03*x)" , "50 + x", 100)
</SCRIPT>
</HTML>
```

В этом примере изображение будет перемещаться по синусоиде с 50 пикселов и горизонтальной скоростью 10 пикселов в секунду. Начальные ординаты графического объекта равны 100 и 50 пикселов по вертикали и горизонтали соответственно.

6.2.2. Перемещение мышью

Элементами перемещения видимых элементов HTML-документа с помощью мыши могут быть графические объекты, кнопки, текстовые области, таблицы, плавающие фреймы и др. Здесь мы займемся перемещением изображений, текстовых областей и плавающих фреймов.

Перемещение графических объектов. Перемещать мышью изображения можно различными способами. Изучим один из них: пользователь пытается перетащить мышью изображение; затем он должен отпустить кнопку мыши и переместить указатель в нужное место (при этом он может удерживать или не удерживать кнопку мыши в нажатом положении); остановившись в нужном месте, пользователь отпускает кнопку мыши или щелкает ею, чтобы прекратить перемещение изображения.

Листинг 6.4. Код перемещения изображения мышью

```
<HTML>
<HEAD><TITLE>Перемещаемое изображение</TITLE></HEAD>
<BODY id = "mybody">
<IMG ID="myimg" SRC = "pict.gif ondragstart = "drag()" style
= "position:absolute; top:10; left:10">
</BODY>
<SCRIPT>
flag = false                              // нельзя перемещать
var id_img = ""
function drag() {
flag = mag
id_img = event.srcElement.id
}
function mybody.onmousemove(){
if (flag){                                // если можно перемещать
document.all[id_img].style.top = event.clientY
document.all[id_img].style.left = event .clientX } }
function mybody.onmouseup(){
flag = false     // нельзя перемещать
}
</SCRIPT>
</HTML>
```

Здесь функция *drag()*, обрабатывающая событие *ondragstart* (попытка перетаскивания), устанавливает переменную-триггер flag и выясняет, кто инициатор события. Значение переменной flag позволяет определить, можно или нельзя перемещать элемент. В данном примере инициатором события может быть только

один элемент. События onmousemove (перемещение указателя мыши) и onmouseup (кнопка мыши отпущена) получает не изображение, а объект тела документа *mybod*y. Здесь не использовалось событие *onclick* для изображения, зарезервировав его для других целей, например для перехода по ссылке. Заметьте также, что в функции *mybody.onmousemove()* при обращении к объекту изображения его идентификатор передается через переменную *id_img*. Поэтому мы используем запись *document.all[id_img*], а не *document.all.idimg*.

Перемещение текстовых областей. В листинге 6.5 приводится пример HTML-документа, в котором можно перемещать текстовые области, созданные с помощью тегов *<TEXTAREA>*. При этом размеры области и шрифта текста определяются в ней в зависимости от значения вертикальной координаты. Так создается эффект перспективы (ближе-дальше). Чем выше, тем дальше, и наоборот. Чтобы приблизить к себе текстовую область, необходимо просто переместить ее вниз. Можно «сложить в стопку» текстовые области, удалить их от себя или, наоборот, приблизить так, чтобы текст стал разборчивым. В отличие от примера с изображениями, здесь разрешение на перемещение происходит по двойному щелчку на текстовой области. Обратите внимание на то, как в функции *resizetext()* определяется элемент, размеры которого следует изменить.

Листинг 6.5. Перемещение текстовых областей, созданных с помощью тегов <TEXTAREA>

```
<HTML>
<HEAD><TITLE>Перемещаемые текстовые области</TITLE>
</HEAD>
<BODY id = "mybody" background = "blue.gif">
<TEXTAREA ID="tl" ondblclick=""drag()" STYLE = "posi-
tion:absolute;
top:10; left: 10; fon-size : large"> Это – первый
текст</TEXTAREA>
<TEXTAREA ID="t2" ondblclick=""drag()" STYLE = "posi-
tion:absolute;
top:100; left:150"> Это – второй текст </TEXTAREA>
```

```
<TEXTAREA ID="t3" ondblclick=""drag()" STYLE = "position:absolute;
top:150; left:250”> Это – третий текст </TEXTAREA>
</BODY>
<SCRIPT>
resizetext()                    /* установка размеров текстовых областей var flag = false */
var id_img = ""
function drag() {
flag = !flag
id_img = event.srcElement.id    /* id элемента, который надо перемещать */
function mybody.onmousemove() {
if (flag){
document.all[id_img].style.top = event.clientY
document.all[id_img].style.left = event.clientX
resizetext() }}                 /* установка размеров текстовых областей */
function mybody.onmouseup() {
flag = false }
function resizetext () {        // установка размеров областей
vary, size, idimg, idtext
for (i =0 ; i < document . all . length; i++) {
if (document.all[i].tagName=='TEXTAREA') {
idtext = document.all[i].id
y = parseInt(document.all[idtext].style.top)
size = Math.min(y, 800)
size = Math.max(size, 60)
document.all[idtext].style.width = size
document.all[idtext].style.height = 0.8*size
document.all[idtext].style.zlndex = y
document.all[idtext].style.fontSize = Math.max(2, y/10)
</SCRIPT>
</HTML>
```

В принципе, ничто не мешает вам создать более сложный алгоритм для функции *resizetext()* изменения размеров текстовой области. Например, можно дополнительно варьировать цвет фона и шрифта.

§ 6.3. Рисование линий

В JavaScript нет специальных встроенных средств для рисования произвольных линий. Если вам потребуется отобразить в окне браузера прямоугольник или горизонтальную линию, то для этого можно воспользоваться HTML-тегами *<TABLE>* и *<HR>* соответственно. А как быть, если нужны наклонная прямая, круг или кривая, заданная уравнением? Например, как изобразить график некоторой зависимости в виде кривой, а не последовательности столбиков?

Идея решения этой задачи довольно проста. Нужно вывести на экран изображение размером 1×1 пиксел, залитое цветом, отличающимся от цвета фона. Это изображение следует разместить несколько раз в соответствии с координатами, которые задаются параметрами позиционирования top и left атрибута *STYLE* тега *<IMG>*. С помощью сценария можно сформировать строку, содержащую теги *<IMG>* с необходимыми атрибутами, а затем записать ее в документ методом *write()*.

В этом параграфе мы рассмотрим сначала задачу рисования прямой линии, а затем – произвольной кривой, заданной выражением с одним аргументом.

6.3.1. Прямая линия

Прежде всего выясним, как нарисовать точку. Из множества таких точек будет состоять любая кривая, которую нам нужно отобразить в окне браузера. В HTML для этого можно использовать следующий тег:

<IMG SRC = "point.bmp" STYLE = "position:absolute;top:y;left:x">

Здесь *point.bmp* – имя графического файла, содержащего один пиксел; *y, x* – числа, указывающие положение графического файла в пикселах. Изображение точки размером 1×1 пиксел можно создать в любом графическом редакторе. Из соображений экономии его лучше всего сохранить в файле формата BMP, а не JPEG или GIF (при малых размерах изображения алгоритмы сжатия неэффективны).

Чтобы задать размеры отображения точки на экране, следует использовать атрибуты *WIDTH* и *HEIGHT* (ширина и высота):

<IMG SRC = "point.bmp" STYLE = "position : absolute; top:y ;left :x" WIDTH=n HEIGHT=n>

Одинаковые значения атрибутов *WIDTH* и *HEIGHT* задают представление точки в виде квадрата размером *n*×*n* пикселов. При этом точка с исходными размерами 1×1 пиксел просто растягивается. Таким образом, имеется возможность задать отображаемые размеры (масштаб) одной точки, а следовательно и определить толщину линии.

Теперь рассмотрим начальную версию функции, которая рисует прямую линию с заданными координатами *x1*, *y1*, *x2*, *y2* ее начала и конца. Кроме координат функция будет принимать числовой параметр *n*, указывающий толщину линии. Вот вариант определения функции:

```
function line(x1, y1, x2, y2 , n){
/* Версия 0 */
/* x1, y1 – начало линии x2 , y2 – конец линии, n – толщина линии */
var clinewidth = " WIDTH=" + n + "HEIGHT=" + n  /* строка для учета толщины */
var xstr = ""            // строка тегов для записи в HTML-документ
var xstr0 = ' <IMG SRC="point.bmp"' + clinewidth + ' STYLE = "positlon:absolute; '
var k = (y2 – y1)/(x2 – x1)          // коэффициент наклона линии
var x = x1                        // начальное значение координаты x
/* Формирование строки, содержащей теги <IMG. . . > : */
while (x <= x2) {
xstr += x s t r 9 + 'top:' + (yl + k* (x – x1)) + ': left:' + x + x+ +
// запись в документ
```

Функция *line()* формирует строку, содержащую теги вывода экземпляров одного и того же графического изображения точки. При этом в цикле изменяются значения параметров *top* и *left* атрибута *STYLE*. Параметром цикла является горизонтальная координата, текущее значение которой присваивается параметру *left*. Таким образом, мы используем горизонтальную развертку линии. Вертикальная координата (top) вычисляется с учетом ее наклона – коэффициента, равного тангенсу угла наклона. Сформированная строка записывается в документ и отображается в окне браузера.

6.3.2. Произвольная кривая

При написании кода функции используем большую часть опыта, полученного при решении задачи рисования прямых линий. Будем считать, что выражение, задающее кривую, содержит переменную, обозначенную строчной латинской буквой *x*. Тогда горизонтальная координата точки кривой вычисляется просто как значение переменной *x*, изменяемой в заданных пределах. Для вычисления вертикальной координаты используем функцию *eval()*, которой передадим в качестве параметра строку, содержащую выражение.

Функция *curve()* для рисования кривой будет принимать следующие параметры: имя графического файла с изображением точки (впрочем, это может быть любое изображение), выражение, задающее кривую, координаты начала линии, количество точек линии, толщину линии и длину штриха (если потребуется штриховая линия).

Листинг 6.6. Код функции curve() для рисования кривой

```
function curve(pict_file,yexpr, x0, y0, t, n. s){ /* Версия 0 */
/* pict_file – имя графического файла
yexpr – выражение с переменной x
x0, y0 – координаты начала кривой
t – количество точек кривой (значений переменной x)
n – толщина линии
s – длина штриха и паузы
*/
if (!yexpr) return null
if (!pict_file) pict_file = "point.bmp"
if (!s) s = 0
if (!t) t = 0
var clinewidth = ''
if (!n)
clinewidth = 'WIDTH=' + n + 'HEIGHT=' + n
var x
xstrG = '<IMGSRO "' + pict_f ile + '"' + clinewidth + STYLE =
"position:absolute;top: '
xstr = ""
var i = 0, draw = true
for(x = 0; x < t; x++) {
if (draw)
xstr += xstrO + (y0 + eval(yexpr)) + '; left:' + (x0 + x)
```

```
if (i > s&&s > 0) {
draw = !draw
i = 0
}
i ++
}
document .write (xstr)      // запись в документ
```

6.3.3. Графики зависимостей, заданных выражениями

Для того, чтобы нарисовать график функции одной переменной в прямоугольной системе координат, необходимо решить, как пересекаются координатные оси: какие квадранты они образуют и сколько. Во-вторых, требуется определить длину каждой из осей, которая, очевидно, зависит от максимального и минимального значений выражения, описывающего кривую. Наконец, следует задать способ оцифровки осей. Код программы, решающей эти задачи, получается значительно объемнее, чем код, обеспечивающий собственно рисование линий. Попробуйте написать его самостоятельно в качестве упражнения. В результате вы получите функцию, с помощью которой сможете рисовать графики различных зависимостей. Не забудьте при этом, что выше мы рассматривали отображение линий в экранной системе координат, в которой вертикальная ось направлена сверху вниз. В обычной практике рисования графиков вертикальную ось направляют, наоборот, снизу вверх.

6.3.4. Графики зависимостей, заданных массивами

Как известно, зависимости между двумя величинами можно задавать в виде таблиц, состоящих из двух столбцов, в одном из которых располагаются данные, соответствующие аргументу зависимости, а в другом – ее значению. Аргументы отображаются на графике вдоль горизонтальной оси, а значения – вдоль вертикальной оси координат. Собственно зависимость обычно представляется либо точками на координатной плоскости, либо ломаной линией, проходящей через эти точки, либо вертикальными прямыми (столбиками). Возможны, конечно, и другие способы графического представления данных (например, в виде круговой диаграммы). Здесь мы остановимся на задаче построения графиков в прямоугольной системе координат в виде ломаной линии.

Таблицы с данными можно представить с помощью одного двухмерного или двух одномерных массивов. Будем считать, что имеем дело с двумя одномерными массивами, в одном из которых содержатся аргументы, а во втором – значения зависимости. Например, в одном массиве содержатся даты, а в другом – соответствующие этим датам среднесуточные (или какие-нибудь другие) значения температуры. Кривую зависимости между температурой и датой можно представить как ломаную линию, состоящую из отрезков прямых линий.

Глава 7
ОБРАБОТКА ДАННЫХ ФОРМ

Такие элементы HTML-документа, как поля ввода данных, текстовые области, переключатели и флажки, раскрывающиеся списки и кнопки, можно объединить в так называемую форму (рис. 7.1). В HTML форма создается с помощью контейнерного тега <*FORM*>, внутри которого располагаются теги элементов этой формы. Однако форма – не просто контейнер, а контейнер и объект, предназначенные главным образом для организации отправки на сервер всех данных, имеющихся в элементах этой формы (например, введенных пользователем). Давным-давно, когда браузеры воспринимали только простой HTML и не работали со сценариями, форма была единственным средством поддержки интерактивности.

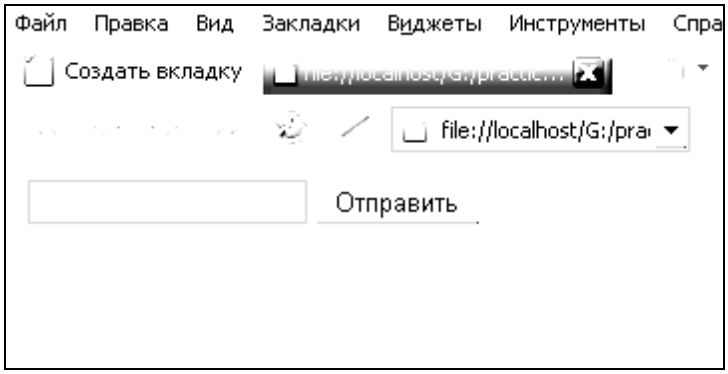

Рис. 7.1. Пример объединения в форму

Как раньше, так и теперь для отправки данных на сервер сценарий не обязателен. Чтобы отправить данные, достаточно в теге <*FORM*> указать атрибут *ACTION*, а в самой форме установить кнопку типа *Submit*. Щелчок на этой кнопке инициализирует отправку данных. Если атрибут *ACTION* не указан или его значение пусто, данные формы не будут отправлены, даже если вы щелкнете на кнопке типа *Submit*. Итак, для отправки данных формы атри-

бут *ACTION* должен иметь некоторое значение. В общем случае это URL-адрес файла или CGI-программы, которая получает и обрабатывает отправленные данные.

Вот пример HTML-документа с формой, содержащей поле ввода данных и кнопку типа *Submit*:

```
<HTML>
<FORM METHOD = POST ACTION = "mailto:mufliu@geras1in.ru"
ENCTYPE="text/plain">
<INPUT NAME = "Сообщение" TYPE = "text" VALUE = "">
<INPUT NAME = "Отправить" TYPE="submit" VALUE =
"Отправить">
</FORM>
</HTML>
```

Если перед отправкой данных формы требуется предварительно их проверить или еще что-нибудь сделать, то для этого необходим сценарий. В следующем примере проверяется, имеется ли символ «@» в поле ввода адреса электронной почты получателя и не пусто ли поле ввода собственно сообщения. Если символа «@» в адресе нет или поле сообщения пусто, то отправка не производится. Сценарий обрабатывает событие onsubmit, возникающее при щелчке на кнопке типа *Submit*.

```
<HTML>
<FORM ID = "myform" METHOD=POST ACTION=""
ENCTYPE="text/plain"
styIe="background:'eЭeOeO1">
Кому:
<INPUT NAME="email_from" TYPE = " text " VALUE = "">
<P>
От к о г о :
<INPUT NAME="email_from" TYPE = " text " VALUE = "">
<P>
Сообщение:<BR>
<TEXTAREA NAME = "Сообщение" TYPE = "text" VALUE = "
<P>
<! Кнопка типа Submit >
<INPUT NAME = "Отправить" TYPE = "submit" VALUE =
"Отправить">
</FORM>
```

```
<SCRIPT>
function myform.onsubmit (){
var noemail = myform.email_to.value.indexOf('@') == - 1
/* присваиваем значение равенства */
var notext = Imyform.Сообщение.value
var xtext = "Письмо не отправлено"
if (noemail | | notext){
event.returnValue = false // отменить отправку
if (noemail)
alert("Неправильный адрес получателя" + xtext)
else
alert("HeT текста сообщения" + xtext)
} else
= " + myform.email_to.value // значение ACTION
}
</SCRIPT>
</HTML>
```

Пример выполнения кода представлен на рис. 7.2.

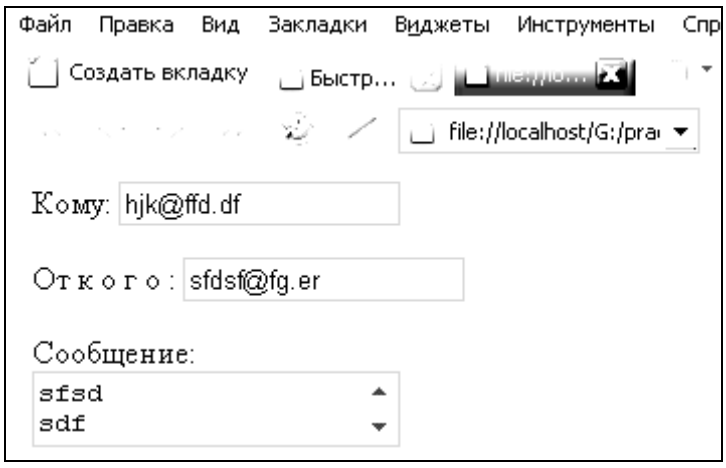

Рис. 7.2. Пример выполнения кода

§ 7.1. Меню

Раскрывающийся список. Простейшее меню можно создать с помощью тегов *<SELECT>* и *<OPTION>*. Обычно такие конструкции называют раскрывающимися списками. Ниже приводится простейший пример использования раскрывающегося списка.

В нем раскрывающийся список задается HTML-кодом, а обработка выбора из этого списка – сценарием. Задача сценария заключается просто в обработке номера выбранного элемента из списка. В примере это вывод окна с соответствующим сообщением. Выбор пользователя из раскрывающегося списка производится щелчком левой кнопкой мыши на элементе списка. При этом свойство *selectedIndex* объекта элемента документа, соответствующего тегу <*SELECT*>, приобретает в качестве своего значения номер выбранного элемента списка (нумерация начинается с 0). Для инициации обработки выбора пользователя здесь служит событие onchange (произошло изменение в выделении элемента списка). Обработка этого события осуществляется функцией *myselectionQ*. Начальное выделение и отображение элемента в раскрывающемся списке задается атрибутом *SELECTED* тега <*OPTION*>.

```
<HTML>
Выберите экзамен, который хотите сдать:
<SELECT NAME="TEST" onchange =
<OPTION>Математика
<OPTION SELECTED>Физика
<OPTION>Биология
<OPTION>Химия
</SELECT>
<SCRIPT>
function myselection() {
var testname, testnumber;
testnumber = document.all .TEST.selectedIndex
if (testnumber == 0)
testname="MaT . анализ"
else{
if (testnumber == 1)
testname="КВаНТОВая физика^
else{
if (testnumber == 2)
testname="Биология"
else
testname = "Органическая химия"
}
alert"Вы будете сдавать: " testname)
}
</SCRIPT>
</HTML>
```

Пример выполнения кода представлен на рис. 7.3.

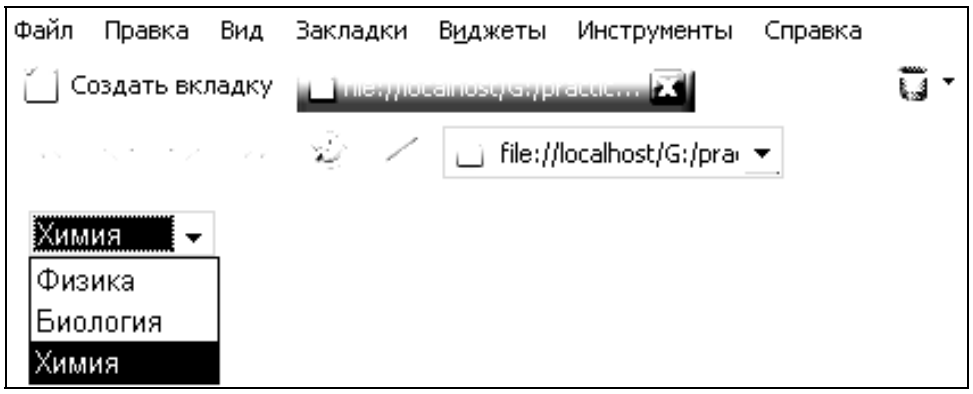

Рис. 7.3. Пример выполнения кода

Поиск в текстовой области. Тексты большого объема, расположенные на веб-странице, обычно снабжают поисковой системой. Внешне она обычно выглядит как поле ввода поискового образа (набора символов, который требуется найти в тексте) и кнопки, щелчок на которой запускает процедуру поиска. В простейшем варианте эта процедура прокручивает текст в окне так, чтобы найденный поисковый образ оказался видимым и выделенным. Пример приведен на рис. 7.4.

Для решения задачи поиска в тексте используется объект *TextRange*. Этот объект просто обеспечивает доступ к текстовой информации, находящейся в объектах, которые соответствуют тегам *<BODY>*, *<TEXTAREA>*, *<BUTTON>* и *<INPUTTYPE = "text">*.

```
<HTML>
<BODY>
<INPUT TYPE = " text " NAME = "WORD" VALUE = "" SIZE = 20 >
<BUTTON onclick = "myfind()">поиск</'BUTTON>
<! Здесь расположен текст, в котором производится поиск >
</BODY>
<SCRIPT>
function myfind() {
ob] = document.body.createTextRange()
obj.findText(WORD.value)
obj.scrollIntoViewO
obj. select()
// создаем текстовую область
```

```
// производим поиск, прокручиваем текстовую область в окне
// выделяем найденное
</SCRIPT>
</HTML>
```

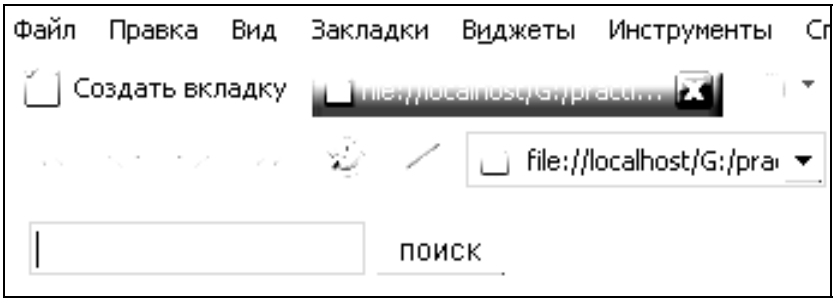

Рис. 7.4. Меню, представленное
в виде пространственно разнесенных частей

Если поиск выполнен неудачно (поисковый образ, введенный в поле ввода, не найден), то ничего не произойдет. А что будет, если пользователь нажал кнопку Поиск, когда ничего не было введено? Появится сообщение об ошибке. Чтобы исключить подобную неприятность, потребуется несколько изменить код программы. А именно следует проверить, не является ли введенный поисковый образ пустым. Ниже приведен вариант кода функции:

```
function myfind(){
if (IWORD.value) return // если поисковый образ пуст, выходим
obj = document. body .createTextRangeO
obj .findText (WORD. value)
obj.scroll I ntoViewO
obj .select()
}
```

§ 7.2. Таблицы и простые базы данных

Таблицы являются наиболее часто используемыми элементами в вебдизайне. Это обусловлено простотой и удобством компоновки документа с помощью тегов таблицы, таких как *<TABLE>*, *<TR>*, *<TD>* и др. Вы разбиваете все пространство окна на прямоугольные ячейки с видимыми или невидимыми границами и

размещаете в них элементы документа (изображения, тексты, ссылки, кнопки, другие таблицы и т. п.). Таким образом, таблицы выполняют роль каркаса документа. До появления CSS они оставались единственным средством точного позиционирования элементов, но и сейчас широко используются как мощный инструмент для понятной организации и эффектного представления информации.

7.2.1. Доступ к элементам таблицы

Таблица как объект документа имеет две коллекции, посредством которых осуществляется доступ к ее содержимому. Первая из них – коллекция строк rows, а вторая – коллекция ячеек cells. Коллекция rows содержит все строки таблицы, включая разделы, соответствующие тегам *<THEAD>* и *<TFOOT>*. Коллекция cells содержит все элементы таблицы, созданные с помощью тегов *<TH>* и *<TD>*. Доступ к элементам коллекций осуществляется либо по индексу, либо по значению атрибута ID в соответствующем теге. Так, для доступа к строке таблицы можно использовать значение ID в теге *<TR>*, а для доступа к ячейке – значение ID в теге *<TH>* или *<TD>*. При использовании индекса (номера) следует иметь в виду, что нумерация начинается с 0. При этом ячейки таблицы нумеруются слева направо и сверху вниз.

Рассмотрим пример простой таблицы, содержащей три столбца и четыре строки.

```
<HTML>
<TABLE ID = "mytab">
<THEAD>Моя таблица</THEAD>
<TH>Фамилия</TH><TH>Имя</TH><TH>Должность</TH>
<TR ID="r1">
<TD>ИВаНОВ</TD><TD>иВаН</TD><TD>реКТор</TD>
</TR>
<TR ID = "r2" >
<TO>Петров</TO><TO>Петр</TO><TO>Заместитель директора</TO>
</TR>
<TR ID="r3">
<TD>СороВа</TD><TD>ЗойКа</TD><TD>СеКреТарша</TD>
</TR>
<TR ID="r4">
```

```
<TO>Федоров</TO><TO>Федор</TO><TO>Водитель</TO>
</TR>
</TABLE>
</HTML>
```

Каждая строка таблицы (элемент коллекции rows), являясь объектом, имеет свою собственную коллекцию ячеек, которая называется cells, так же как и коллекция всех ячеек таблицы. Однако нумерация ячеек в этой коллекции происходит в пределах одной строки начиная с 0. Например, в случае трехстолбцовой таблицы ячейки из коллекции cells для одной строки имеют индексы 0, 1 и *2*.

7.2.2. Генерация таблиц с помощью сценария

Один из неприятных моментов в создании таблиц посредством тегов HTML состоит в большом количестве этих тегов и необходимости тщательно следить за правильной их расстановкой. При этом коррекция содержимого ячеек больших таблиц оказывается весьма хлопотным делом. Облегчить работу в ряде случаев помогает хранение содержимого ячеек в массивах и генерация таблицы с помощью сценария. В следующем примере предполагается, что содержимое строк таблицы хранится в отдельных массивах. С помощью операторов цикла формируется строка, содержащая теги таблицы, а затем эта строка записывается в HTML-документ.

```
<HTML>
<SCRIPT>
/* Массив заголовков столбцов */
ah = new Array ("Фамилия" ,"Имя","Должность"
/* Массив данных */
ad = new Array()
ad[0] = new Array( "Иванов", "Иван","директор")
ad[1] = new Array( "Петров","Петр","зам")
ad[2] = new Array ("Сидорова","Эллочка","секретарь")
ad[3] = new Array ( "Федоров", "Федор","водитель")
strtab = "<TABLE>"
/* Формируем заголовки столбцов
for( i = 0 ; i<ah.length; i++){
strtab + = "<TH>" + ad[i] + </TH>
}
```

```
/* Формируем строки таблицы */
for(i=0; i<ad.length; i++) {
strtab+= "<TR>"
f o r ( j = 0 ; ]'<ad[i].l e n g t h ; j++){
strtab+="<TD>"+ad[i][j]+"</TD>"
}
strtab+="</TR>"
|
strtab+= "</TABLE>"
document.write(strtab)
</SCRIPT>
</HTML>
```

7.2.3. Сортировка данных таблицы

Рассмотрим сортировку строк таблицы по значениям того или иного столбца. В нашем примере для сортировки необходимо просто щелкнуть на заголовке нужного столбца таблицы. Для этого кроме элемента управления STD и таблицы нам понадобится сценарий, содержащий функцию, обрабатывающую щелчок путем модификации тега <*OBJECT*> (листинг 7.1). Здесь использован следующий хитрый прием. Сначала база данных показывается без всякой сортировки. Сохраняется блок тегов от <*OBJECT*> до </*OBJECT*>, за исключением последнего, в переменной *obj*. При щелчке на заголовке таблицы вызывается функция сортировки *sort(field)*, которой передается имя поля. Эта функция дописывает к значению переменной *obj* строку, содержащую теги параметров, отвечающие за сортировку. Далее функция *sort()* заменяет в HTML-коде теги <*OBJECT*> на те, которые содержатся в переменной *obj*. Это делается с помощью свойства *outerHTML*. При этом элемент управления STD заново инициализируется с новыми параметрами. В результате таблица перерисовывается в соответствии с новым порядком следования строк.

Листинг 7.1. Код сценария сортировки данных таблицы

```
<HTML>
<OBJECT ID = "mydbcontrol"
CLASSID = "CLSID:333C7BC4-460F-11D0-BC04-0080C7055A83">
<PARAM NAME "FieldDelim" VALUE="|">
<PARAM NAME "DataURL" VALUE = "mydb.txt">
```

```
<PARAM NAME "UseHeader" VALUE = true>
<PARAM NAME "SortColumn" VALUE"Фамилия">
<PARAM NAME "SortAscending" VALUE = 1>
</OBJECT>
<TABLE DATASRC = tfmydbcontrol BORDER = 5>
<THEAD>
<TH onclick = "sort('Фамилия')">Фамилия сотрудника</TH>
<TH onclick = "sort('Имя')">Имя/TH>
<TH>Фото</TH>
</THEAD>
<TR>
<TD><SPAN DATAFLD = "Фамилия" ></ SPAN></TD>
<TD><SPAN DATAFLD = " Имя " >< / SPAN></TD>
<TD><SPAN DATAFLD = "Портрет" DATAFORMATAS =
"html" > </SPAN></TD>
</TR>
</TABLE>
<SCRIPT>
// Сохраняем теги объекта STD в переменной
var x = document.all.mydbcontrol.innerHTML /* теги, вложенные в
<object>*/
var obj = '<OBJECT ID = "mydbcontrol" CLASSID =
"CLSID:333C7BC4-460F-11D0-BC04-0080C7055A83">' + x
function sort(field){ /* сортировка по значениям столбца field */
var y = document.all.mydbcontrol
y.outerHTML = obj + '<PARAMNAME = "SortColumn"
VALUE="' + field +
'"></OBJECT>'
}
</SCRIPT>
</HTML>
```

В этом примере сортировать данные можно только по первым двум текстовым полям. Можно в качестве упражнения усовершенствовать этот код. Например, сделать так, чтобы при двойном щелчке на заголовке столбца таблицы сортировка происходила в противоположном порядке. Для этого необходимо не устанавливать параметр *<PARAM NAME="SortAscending" VALUE=0>*.

7.2.4. Фильтрация данных таблицы

Для организации фильтрации данных из таблицы по заданному критерию используется подход, аналогичный рассмотренному в предыдущем подпараграфе. В приведенном ниже примере (листинг 7.2) в окно браузера выводится таблица с данными и элементы, с помощью которых можно задать поле (столбец) и значение (образец) для формирования простого условия фильтра вида: *имя_поля = значение*.

Имя поля выбирается из раскрывающегося списка, а значение вводится с клавиатуры. Для установки фильтра следует щелкнуть на кнопке Применить.

Если поле ввода значения пусто, то при щелчке на этой кнопке фильтр выключится, а в таблице будут отображены все имеющиеся записи. Заметим, что результат фильтрации в данном примере не зависит от регистра, в котором пользователь ввел значение.

Также, как и при сортировке, мы сохраняем в переменной часть информации о параметрах элемента, а затем модифицируем ее, чтобы учесть условие фильтра. Для установки фильтра используется свойство outerHTML.

Листинг 7.2. Код для фильтрации данных таблицы

```
<HTML>
<H3>Фильтр:</H3>
Поле
<! Раскрывающийся список имен полей >
<SELECT NAME = "FLD">
<OPTION value = "Фамилия">Фамилия
<OPTION value = "Имя">Имя
</SELECT>
<BR>
Значение
<! Поле ввода значения для фильтра и кнопка>
<INPUT NAME = "IMP" VALUE = " TYPE = "text">
<P>
<BUTTON onclick = "filter() ">Применить</Button>
<HR>
<! Элемент управления STD>
<BJECT ID = "mydbcontrol"
```

```
CLASSID = "CLSID:333C7BC4-460F- 11D0-BC04-0080C7Q55A83">
<PARAM NAME = " FieldDelim" VALUE="|">
<PARAM NAME = "DataURL" VALUE = " mydb.txt" >
<PARAM NAME = "UseHeader" VALUE = true>
</OBJECT>
<! Таблица для вывода данных>
<Table DATASRC = tfmydbcontrol border = 5>
<THEAD>
<TH>Фамилия</TH>
<TH>Имя</TH>
<TH>Портрет</TH>
</THEAD>
<TR>
<TD><SPAN DATAFLD = "Фамилия"></SPAM></TD>
<TD><SPAN DATAFLD = "Имя" ></SPAN></TD>
<TD><SPAN DATAFLD = "Портрет" DATAFORMATAS="html">
<SPAN></TD>
</TR>
</TABLE>
<SCRIPT>
/* сохраняем основные параметры элемента STD в перемен-
ной obj */
var obj = "<OBJECT ID = 'mydbcontrol'
obj+ = "CLASS ID='CLSID:333C7BC4-460F-11D0-BC04-
0080C7055A83'>"
"<PARAM NAME='FieldDelim' Value = ' ) ' > "
obj+= "<PARAM NAME='DataURL' Value = 'mydb.txt' > "
obj+= "<PARAM NAME=' UseHeader' value = true> "
function filter() { // установка фильтра
var cpar = " // переменная для хранения параметров фильтра
if (INP.value){ // если введено значение
cpar+= "<paramname='FiIterValue' value = ' " + I N P . v a l u e +" ' >
cpar+= "<param name='FiIterCriterion' value = '='
"<PARAM NAME='CaseSensitive' VALUE = false>"
}
L = obj + cpar + "</OBJECT>"
}
</SCRIPT>
</HTML>
```

Пример приведен на рис. 7.5.

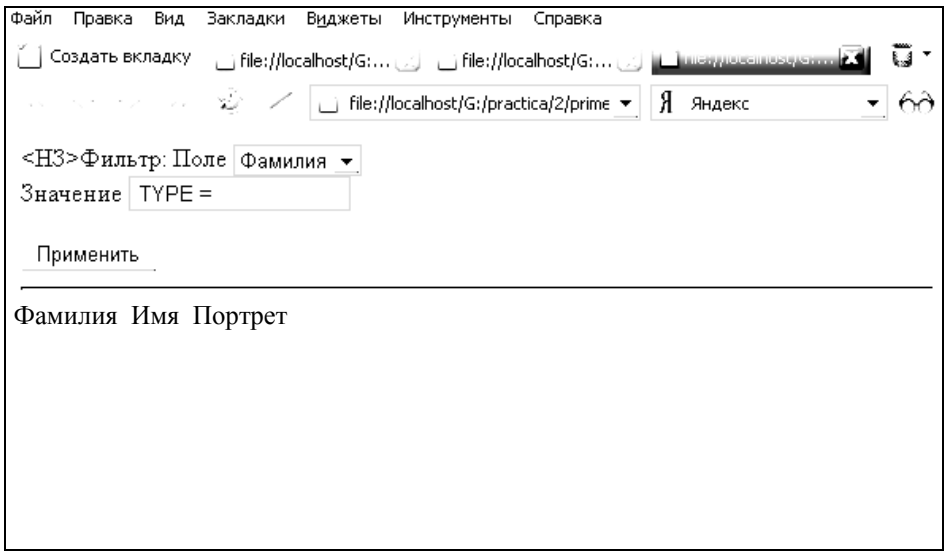

Рис. 7.5. Вставка HTML-документа в таблицу

Документ из внешнего HTML-файла можно вставить в таблицу, находящуюся в текущем документе и управляемую элементом STD. Эта возможность основана на том, что HTML-файл является обычным текстовым файлом. Поскольку в первой строке этого файла указывается тег *<HTML>*, мы можем использовать его в качестве имени поля базы данных. Таким образом, мы рассматриваем вставляемый HTML-файл как базу данных с единственным полем (столбцом). Основное требование к содержимому HTML-файла: любой контейнерный тег должен полностью размещаться в одной строке этого файла.

Рассмотрим сначала пример вставки в таблицу одного HTML-документа:

```
<HTML >
<! Элемент управления >
<OBJECT ID = 'mydbcontrol'
CLASSID = 'CLSID:333C7BC4-460F-11D0-BC04-0080C7055A83I>
<PARAM NAME = 'FieldDelim ' VALUE='|'>
<PARAM NAME = 'DataURL' VALUE='mydocum.htm'>
<PARAM NAME = ' UseHeader ' VALUE=true>
</OBJECT>
<TABLE DATASRC = # mydbcontrol WIDTH = 350>
```

```
<TR>
<TD><SPAN DATAFLD = '<HTML>' DATAFORMATAS = ' html ' ></ SPAN></TD>
</TR>
</TABLE>
</HTML>
```

Здесь предполагается, что в документе тег *<HTML>* записан прописными буквами. Если же он записан строчными буквами, то это должно быть отражено и в значении атрибута *DATAFLD = "<html>"*. Очевидно, можно задать атрибуты оформления таблицы (рамок, выравнивания, цвета и т. п.), а также стилевые параметры позиционирования.

Теперь рассмотрим случай вставки нескольких HTML-документов. В листинге 7.3 приведен сценарий, который это выполняет.

Листинг 7.3. Код сценария вставки нескольких HTML-документов

```
<SCRIPT>
/* Вставка HTML-документов */
/* Массив имен (URL) вставляемых HTML-файлов */
var aurl = new ArrayO
aurl[0] = "html1.htm"
aurl[1] = "html2.htm"
var tabwidth=800; /* ширина таблицы, в которой будут показаны HTML-документы */
/* Массив частей строки, определяющей элемент управления STD */
var xobject = new ArrayO
xobject[0] = <O'BEJC'TD Idojb=i" //id
xobject[l] = '" CLASSID="CLSID:3
xobject[l] += '<PARAMNAME = " FieldDelim " VALUE = " | " > r
xobjectfl] += '<PARAMNAME = "UseHeader" VALUE = time > '
xobject[l] += '<PARAMNAME = "CharSet" VALUE = "windows-1251">'
xobject [1] += '<PARAMNAME = "DataURL" VALUE =" '
xobject[2] = <''></OBJECT>'
/* Строка тегов таблицы */
var xtab = new ArrayO
xtab[0] = '<TABLE WIDTH=' + tabwidth + ' DATASRC = #'
```

```
xtab[l] = 'XTRXTDXSPANDATAFLD="<ritml>" DATAFORMA-
TAS= " html" > '
xtab[l] += '</SPAN></TD></TRx/TABLE>'
var sobj = "";
for (i = 0 ; i < aurl.length; i++) {
sobj += xobject[0] + i + xobjectll] + aurl[i] + xobject[2] + xtab[0] +
[idobj ' + i + xtabfl]
}
document.write(sobj)
</SCRIPT>
```

Для пробы возьмем следующие два простых HTML-документа:

Файл htmll.htm:

```
<HTML>
<H3>Документ 1</H3>
<image src="folder.gif">Это – просто картинка
<a hn Описание баз данных в текстовых файлах</a>
<button onclick="alert('Привет!! о – кнопка
</HTML>
```

Файл html2.htm:

```
<HTML>
<H3>Документ 2</H3>
<1>Это – содержание документа из второго файла.<1>
В нем только этот текст
</HTML>
```

§ 7.3. Защита веб-страниц с помощью пароля

Если необходимо защитить сайт или его отдельную страницу с помощью пароля, то главное требование – обеспечить сохранность пароля, а также адреса защищаемой страницы. Очевидно, они не должны фигурировать в HTML-документе и сценарии в явном виде. Достаточно простой и надежный способ решения этой задачи – совпадение пароля с именем (без расширения) какого-нибудь файла,

расположенного на сервере. В этом случае к слову, введенному пользователем, следует добавить расширение имени файла и проверить, существует ли он на сервере. Если файл существует, то введенный пользователем пароль верен, и можно открыть доступ к документу. В противном случае будет отказано в доступе. Обратите внимание, что имя файла в явном виде не упоминается. Проверку существования файла можно выполнить с помощью метода *FileExistsQ* объекта файловой системы (FileSystemObject). Однако использование этого объекта в сценарии для браузера сопряжено с появлением неприятного сообщения о том, что страница содержит элемент *ActiveX*, который может представлять опасность. При этом пользователю предлагается принять решение о выполнении программы. Если он откажется от ее выполнения, то не получит доступа к защищаемой странице.

Другой вариант решения задачи защиты с помощью пароля тоже основан на совпадении пароля с именем файла. Однако в этом варианте используется не объект файловой системы, а безопасный элемент *ActiveX* управления текстовыми базами данных. Файл с именем, совпадающим с паролем, является текстовым файлом, содержащим базу данных с двумя столбцами (полями) и двумя строками (записями). В первой строке записаны идентификаторы полей базы данных, между которыми указан символ-разделитель. Во второй строке через такой же разделитель тоже записаны всего лишь два слова – имя этого текстового файла без расширения и имя HTML-файла или URL-адрес страницы перехода. Пусть поля базы данных имеют идентификаторы password и myspecurl. Тогда содержимое текстового файла с именем, например, *myspecfile.txt* должно иметь следующий вид:

password]myspecurl
myspecfile|http://www.myweb.ru/mypage.htm

Здесь во второй строке во втором ее поле указан гипотетический URL-адрес страницы, на которую должен перейти браузер, если пользователь введет правильный пароль.

В следующем примере в HTML-документе размещается текстовое поле ввода и кнопка, щелчок на которой обрабатывается сценарием (листинг 7.4). Сценарий создает элемент управления и невидимую таблицу, загружая в нее базу данных. В этой таблице всего две ячейки: одна с паролем, а другая с адресом перехода. Далее слово, введенное пользователем, сравнивается со словом в

таблице. Если они совпадают, то происходит переход на страницу с указанным адресом. В противном случае выдается сообщение о том, что введенный пароль неверен.

Листинг 7.4. Сценарий защиты HTML-документа с помощью пароля

```
<H3>Только для членов клуба</H3>
Введите пароль:
<INPUT ID = "pwl" TYPE="text" VALUE="">
<BUTTON ID = "pwenter">BBOfl</BUTTON>
<BID = "dbelem"></B>
<SCRIPT>
function pwenter.onclick() {
if // если поле ввода пароля пусто
return
dbstr = '<OBJECT ID = "mydbcontrol" ' +
'CLASSID="CLSID:333C7BC4-460F-11D0-BC04-0080C7055A83">' +
'<PARAMNAME= "DataURL" VALUE = '" + document. all.pwl
.value+ '.txt"> ' +
'<PARAMNAME="FieldDelim" VALUE="|">' +
'<PARAMNAME = "UseHeader" VALUE = +
'<TABLE STYLE = "visibility:hidden"><TR><TD>' +
<INPUT ID = "pw2" TYPE = "text" DATASRC = "#mydbcontrol"' +
'DATAFLD= "password"></TD><TD>' +
ID = ' jrl" TYPE = "text" DATASRC = "tfmydbcontrol" ' +
'DATAFLD = "myspecurl"x/TD></TRx/TABLE>'
/* вставляем элементы БД
в документ */
setTimeout ("validationO", 1000) // задержка
}
validationO { // проверка правильности и переход,
if (document.all.pwl.value == document.all.pw2.value) { /*если
пароль верен */
document.all.pwl.value = ""
= // переход,
}else // если пароль не верен
alert("Пароль не верен!")
</SCRIPT>
</HTML>
```

§ 7.4. Взаимодействие с Flash-мультфильмами

Приложения, созданные в системе Macromedia Flash версий 5.0 и 6.0 (MX), могут взаимодействовать со сценариями на JavaScript. А именно, они могут получать данные из сценария, написанного на языке JavaScript, и использовать их некоторым образом с помощью своего сценария, написанного на языке ActionScript. С другой стороны, сценарии на ActionScript могут использовать сценарии (функции) на JavaScript. Чтобы организовать такое взаимодействие, необходимо прежде всего вставить приложение Flash (мультфильм, ролик, клип) в HTML-документ. Это делается с помощью контейнерного тега <OBJECT>, инициирующего Flash-проигрыватель и загружающего в него мультфильм. В IE5+ этот проигрыватель уже встроен.

§ 7.5. Передача данных из JavaScript в ActionScript

В качестве примера передачи данных из JavaScript в Action-Scipt рассмотрим задачу отображения текста в окне Flash-мультфильма. Используя большие изобразительные возможности Flash, можно создать окно произвольной формы, а не только прямоугольное, как в HTML. В нашем примере HTML-документ содержит Flash-проигрыватель с загруженным в него мультфильмом, поле ввода данных и сценарий, который передает во Flash-мультфильм содержимое поля ввода. При этом содержимое поля ввода данных (элемента *<INPUT>*) отображается в окне Flash-мультфильма. В листинге 7.5 приведен соответствующий HTML-код.

Листинг 7.5. Пример передачи данных из JavaScript в ActionScipt

```
<HTML>
<HEAD>
<meta http-equiv = Content-Type content = "text/html; charset =windows-1251">
<TITLE>Передача из JavaScript в Flash</TITLE>
</HEAD>
```

```
<BODY BGCOLOR = "#e0e0e0">
<! Flash-проигрыватель >
<OBJECT CLASSID = " c l s i d : D27CDB6E-AE6D-llcf-96B8-
444553540000"
CODEBASE ="http://download.fflacromedia.com/pub/shockwave
/cabs/flash/swflash.cab#version = 6,0,0,0"
WIDTH = "287" HEIGHT = "261" id = "myflash" ALIGN="">
<PARAM NAME = movie VALUE = "myflash.swf">
<PARAM NAME = quality VALUE=high>
<PARAM NAME = wmode VALUE = transparent
<PARAM NAME = bgcolor VALUE=#FFFFFF>
<EMBED src = "myflash.swf" quality=high wmode=transparent
bgcolor =#FFFFFF WIDTH = "287" HEIGHT = "261" NAME =
"myflash" ALIGN = ""
TYPE = "application/x-shockwave-flash"
PLUGINSPAGE = "http://www.macromedia.com/go/getflashplayer">
</EMBED>
</OBJECT>
<P>
Введите текст в поле, и он отобразится в мультфильме<BK>
<INPUT TYPE = "text" NAME = "inputtext" SIZE = 40 onchange
= getit()>
</BODY>
<SCRIPT>
function getit(){
document.myflash.SeW ariable("inFlash", inputtext.value)
}
</SCRIPT>
</HTML>
```

Здесь с помощью тега <*OBJECT*> в браузер загружается элемент управления ActiveX, соответствующий Flash-проигрывателю версии 6.0 (MX), хотя для воспроизведения нашего мультфильма достаточно и версии 5.0. В этот проигрыватель, в свою очередь, загружается мультфильм из файла myflash.swf. Атрибут *CODEBASE* тега <*OBJECT*> содержит в качестве значения URL-адрес проигрывателя, если он не встроен в браузер пользователя. Заметим, что в IE5+ Flash-проигрыватель уже встроен. Вложенный контейнерный тег <*EMBED*> используется только ради браузера Netscape Navigator. Далее в HTML-документе с помощью тега <*INPUT*>

задается поле ввода данных. При изменении значения этого поля (событие onchange) вызывается функция *getit()*, определенная в сценарии JavaScript. В этой функции с помощью метода *SetVariable()* переменной *inFlash* присваивается значение поля ввода данных inputtext.value. Этого вполне достаточно, чтобы передать значение поля ввода данных в мультфильм. Метод *SetVariable()* является методом объекта Flash-проигрывателя, который в данном примере имеет *ID = "myflash"*. Имя переменной *inFlash* (произвольное) задается как один из параметров Flash-мультфильма. Среди прочих параметров мультфильма отметим еще *<PARAM NAME = wmode VALUE = transparent>*, который задает прозрачность фона занимаемой им прямоугольной области. Это простой и весьма эффективный способ согласования содержимого Flash-мультфильма с общим дизайном веб-страницы. Теперь рассмотрим, как сделать Flash-мультфильм, воспринимающий и отображающий данные из HTML-документа с помощью сценария, написанного на языке JavaScript. В пакете Macromedia Flash MX создадим простой мультфильм, содержащий два слоя. В первом, фоновом слое нарисуем окно. Это может быть любая фигура. В нашем примере используется инструмент Овал, чтобы нарисовать эллипс. Затем с помощью инструмента Subselection (белая стрелка) преобразуется эллипс в некую причудливую форму, напоминающую облако. Применяем к ней заливку радиальным градиентом. Далее необходимо создать второй слой, расположенный над первым. В этом слое в пределах «облака» с помощью инструмента Text создается текстовое поле, параметры которого определяются в палитре Properties (Свойства). Эта палитра во Flash MX еще называется Инспектором свойств. Сначала зададим тип текстового поля – Input Text (Ввод текста). Затем в поле Var (Переменная) введем имя переменной, принимающей текст, который должен отображаться в текстовом поле. В рассматриваемом примере выбрано имя переменной nFlash. Это важный момент, поскольку именно эта переменная указывается в сценарии JavaScript как первый параметр метода SetVariableQ. Наконец, с помощью других органов управления палитры Properties задаем дополнительные параметры текстового поля. В частности, выбираем из раскрывающегося списка имя шрифта, режим отображения текста Multiline (Многострочный) и др.

§ 7.6. Вызов сценария JavaScript из сценария ActionScript

Язык ActionScript для создания сценариев Flash-мультфильмов – достаточно мощное средство программирования, поэтому он может обойтись и без JavaScript. Тем не менее существует возможность вызывать фрагменты JavaScript-кода из сценария, написанного на ActionScript. Это удобно в тех случаях, когда у вас уже созданы программы на JavaScript. Например, зачем писать код функции представления чисел словами на ActionScript, если он уже написан на JavaScript.

Вызов функции JavaScript из сценария ActionScript осуществляется следующим образом:

getURL("javascript: имя_функции(параметры)")

В рассматриваемом примере Flash-мультфильм содержит три кнопки, щелчки на которых обрабатываются функциями, написанными на JavaScript. Если уж быть точным, то щелчок на кнопке в мультфильме обрабатывается сценарием на ActionScript, который вызывает функцию на JavaScript.

HTML-документ содержит тег *<OBJECT>*, загружающий Flash-проигрыватель вместе с мультфильмом, и сценарий, в котором определены три функции: открытия нового окна, вывода диалогового окна с сообщением и закрытия окна. В листинге 7.6 приведен соответствующий код.

Листинг 7.6. Вызов сценария JavaScript из сценария ActionScript

```
<HTML>
<HEAD>
<meta http-equiv = Content-Type content = "text/html;
charset = windows-1251">
<T1T1E>Вызов JavaScript из ActionScript</TITLE>
</HEAD>
<BODY BGCOLOR = "#e0e0eO">
<! Flash-проигрыватель >
<OBJECT CLASSID = "clsid:D27CDB6E-AE6D-llcf-96B8-
444553540000"
```

```
CODEBASE = "http://download.macromedia.com/pub/shoekwave/
cabs/flash/
swflash.cab#version = 6,6,0,0" WIDTH = "400" HEIGHT = "100" id =
"myflash" ALIGN= "">
<PARAM МАЙЕ = movie VALUE = "myflash.swf">
<PARAM NAME = quality VALUE=high>
<PARAM NAME = wmode VALUE = transparent^
<PARAM NAME = bgcolor VALUE=#FFFFFF>
<EMBED src = "myflash.swf" quality=high wmode=transparent
bgcolor =
#FFFFFF WIDTH = "287" HEIGHT = "261" NAME = "myflash"
ALIGN = '
TYPE = "application/x-shockwave-flash" PLUGINSPAGE =
"http://www.macromedia.com/go/getflashplayer">
</EMBED>
</OBJECT>
</BODY>
<SCRIPT>
var mywindow
function newwindow(myURL, mywidth, myheight){ /* открытие
нового окна */
mywindow = wi ndow.open(myURL,'saple' , "toolbar=no, bar=no, l
, resizable=no, width=" + mywidth +
, height=" + myheight + ", top=0, left=0")
I
function message(message){ // вывод сообщения
alert(message)
I
function closewindowO{ // закрытие окна
mywindow
mywindow.close()
}
</SCRIPT>
</HTML>
```

Во Flash-редакторе создадим простой мультфильм, содержащий три кнопки. Для этого можно воспользоваться встроенной библиотекой, содержащей множество уже готовых элементов управления. Каждой кнопке назначим следующие действия (сценарии):

Кнопка 1 (открытие окна и загрузка в него файла temp.htm):

```
on(release){
getURL("javascript: newwindow('temp.htm' , 550, 250)")
}
```

Кнопка 2 (вывод сообщения):

```
on(release){
getURL("javascript: message('Привет из Flash')")
}
```

Кнопка 3 (закрытие окна):

```
on(release){
getURL("javascript: closewindowO")
}
```

Исходный файл созданного мультфильма сохраним под именем myflash.fla. Это необходимо, если нам потребуется в дальнейшем модифицировать его. Для вставки мультфильма в HTML-документ необходимо создать файл в формате SWF, то есть файл myflash.swf. Этой цели служит операция публикации мультфильма Publish, которая выбирается из меню File (Файл) в главном окне пакета Flash. Параметры публикации можно задать по команде File > Publish Settings (Файл > Настройка публикации). В соответствующем диалоговом окне можно задать имя swf-файла, версию Flash-проигрывателя и другие параметры.

Практические задания к главам 6–7

Задание 1

Напишите скрипт «конвертер температур». Это полезно как в обычной жизни, так и для решения задач по физике.

Предлагаемый вариант

```
<form name="converter">
<pre> Число значащих цифр: <input name="precision" value="8" maxlength="12" onkeyup="UpdateF(0)">
Цельсии: <input name="celsius" maxlength="12" onkeyup="UpdateF(1)">
```

```
Кельвин: <input value="NaN" name="kelvin" maxlength="12" onkeyup="UpdateF(2)">
Фаренгейт: <input value="NaN" name="fahrengeit" maxlength="12" onkeyup="UpdateF(3)">
</pre>
</form>
<script language="javascript">
function UpdateF(n)
{
var p = eval(window.document.converter.precision.value);
var c = window.document.converter.celsius;
var k = window.document.converter.kelvin;
var f = window.document.converter.fahrengeit;
if ( n == 0) {
      if (c.value) c.value = eval(c.value).toPrecision(p);
      if (k.value) k.value = eval(k.value).toPrecision(p);
      if (f.value) f.value = eval(f.value).toPrecision(p);
};
if ( n == 1 ) {
      k.value = (eval(c.value) + 273.15).toPrecision(p);
      f.value = (1.8 * eval(c.value) + 32).toPrecision(p);
};
if ( n == 2 ) {
      c.value = (eval(k.value) - 273.15).toPrecision(p);
      f.value = (1.8 * (eval(k.value)) - 459.67).toPrecision(p);
};
if ( n == 3 ) {
      c.value = ( (eval(f.value) - 32) / 1.8 ).toPrecision(p);
      k.value = ( (eval(f.value) + 459.67) / 1.8).toPrecision(p);
};
}
-->
</script>
```

Задание 2

Создайте переменную-триггер (так называемый флаг), принимающий одно из двух возможных значений. По текущему значению флага сценарий может определить, какое именно из двух изображений следует отобразить. После смены изображения необходимо изменить и значение флага. Далее приведен вариант кода (рис. 7.6).

```
<HTML>
<IMG ID = "myimg" SRC = '1_1_files/pict1.jpg' onclick = "imgchange()">
<SCRIPT>
var flag=false          // флаг (триггер)
function imgchange() {  // обработчик щелчка на изображении
if (flag) document.all.myimg.src = "1_1_files/pict1.jpg"
else document.all.myimg.src = "1_1_files/pict2.jpg"
flag=!flag //изменяем значение флага на противоположное
}
</SCRIPT>
</HTML>
```

Рис. 7.6. Пример кода

Идентификаторы ID тегов <*IMG*> задайте некоторым регулярным образом (например "10", "11", "12", ...). Так часто поступают при использовании массивов. Создайте еще два массива, содержащих имена графических файлов: один для исходных изображений, а другой – для замещающих. Наконец, HTML-документ с изображениями сгенерируйте с помощью сценария. Для этого сначала сформируйте строку, содержащую теги <*IMG ...*>, а затем запишите ее в документ.

Задание 3

Условие: три элемента, задающие кнопки, находятся в контейнере формы <*FORM*>. К этому контейнеру привязываются обработчики событий onmouseover (наведение указателя мыши) и onmouseout (удаление указателя мыши). Таким образом, инициатором (получателем) этих событий может быть любой элемент формы (в нашем примере – любая из трех кнопок). В обычном состоянии кнопки имеют серый цвет, заданный 16-м кодом a0a0a0. При наведении указателя мыши цвет кнопки становится желтым (yellow) (рис. 7.7).

```
<HTML>
<STYLE>
mystyle {font-weight: bold ; background-color: a0a0a0}
</STYLE>
 <FORM onmouseover = "colorchange('yellow')" onmouseout = "colorchange('a0a0a0')">
<INPUT TYPE = "BUTTON" VALUE = "Первая" CLASS = "mystyle" onclick =
"alert('Вы нажали кнопку') ">
<INPUT TYPE = "BUTTON" VALUE = "Вторая" CLASS = "mystyle" onclick =
"alert('Вы нажали кнопку 2')">
<INPUT TYPE = "BUTTON" VALUE = "Третья" CLASS = "mystyle" onclick =
"alert('Вы нажали кнопку 3')">
</FORM>
<SCRIPT>
function colorchahge(color) {                    // изменение цвета кнопок
if(event.srcElement.type == "button")
event.srcElement.style.backgroundColor = color;
}
</SCRIPT>
</HTML>
```

Рис. 7.7. Пример кода

Задание 3

Создайте прямоугольную рамку, окаймляющую некий текст, которая периодически изменяет цвет. Рамка создается тегами одноячеечной таблицы с заданием нужных атрибутов и параметров стиля. Далее необходимо создать функцию, изменяющую цвет рамки таблицы на другой, и передать ее в качестве первого параметра методу setInterval(). Второй параметр этого метода задает период в миллисекундах, с которым вызывается функция, указанная в первом параметре (рис. 7.8).

```
<HTML>
<TABLE ID = "mytab" BORDER=1 WIDTH=150 style = "border:10 solid:yellow">
<TR><TD>Мигающая рамка</TD></TR>
</TABLE>
<SCRIPT>
function flash() {              //изменение цвета рамки,
if (!document.all)              //если в документе ничего нет
return null;
if (mytab.style.borderColor=='yellow')
        mytab.style.borderColor='red'
else
        mytab.style.borderColor='yellow';
}
setInterval("flash()",500); // мигание рамки с интервалом 500 мс
</SCRIPT>
</HTML>
```

Рис. 7.8. Пример кода

Задание 4

Просмотрите главу 7 и, исходя из данной в ней теории, составьте форму для отправки сообщения.

Задание 5

Аналогично заданию 4 составьте скрипт, защищающий страницу паролем.

Задание 6

Просмотрите готовые скрипты и составьте из них связанные между собой страницы так, чтобы каждая страница была не перегружена кодом (не стоит все помещать на одну страничку), но она и не была пустой. Готовый проект может быть шаблоном для вашего будущего сайта.

ПРИЛОЖЕНИЕ

Следующие инструкции Вам объяснят, как включить JavaScript в Вашем браузере. Если Ваш браузер ниже не указан, мы советуем Вам проконсультироваться в центре технической поддержки этого браузера.

Internet Explorer (6.0)

1. Выберите «Сервис» из верхнего меню.
2. Выберите «Свойства обозревателя».
3. Кликните на закладку «Безопасность».
4. Кликните на кнопку «Другой».
5. Прокрутите вниз до раздела «Сценарии».
6. В разделе «Активные сценарии», выберите «Включить» и кликните OK.

Netscape Navigator (7.1)

1. Выберите «Правка» из верхнего меню.
2. Выберите «Настройки».
3. Выберите «Дополнительно».
4. Выберите «Скрипты и плагины».
5. Поставьте галочку рядом с «Включить JavaScript» и кликните OK.

Mozilla Firefox (1.0)

1. Выберите «Инструменты» из верхнего меню.
2. Выберите «Настройки».
3. Выберите «Веб Функции» из левой навигационной панели.
4. Поставьте галочку рядом с «Включить JavaScript» и кликните OK.

Apple Safari (1.0)

1. Выберите «Safari» из верхнего меню.
2. Выберите «Настройки».
3. Выберите «Безопасность».
4. Поставьте галочку рядом с «Включить JavaScript».

Помните, что обновление браузера или установка новых программ безопасности может повлиять на Ваши настройки JavaScript.

ЛИТЕРАТУРА

1. Кенцл, Т. Форматы файлов Internet / Т. Кенцел; пер. с англ. – СПб: Питер, 1997. – 320 с.

2. Фролов, А. В. Сервер Web своими руками. Язык HTML, приложения CGI и ISAPI, установка серверов Web для Windows / А. В. Фролов, Г. В. Фролов. – М.: Диалог-МИФИ, 1998. – 288 с.

3. Мейнджер, Дж. JavaScript: основы программирования / Дж. Мейнджер; пер. с англ. – Киев: Издат. группа BHV, 1997. – 512 с.

4. Фролов, А. В. Сценарии JavaScript в активных страницах Web / А. В. Фролов, Г. В. Фролов. – М.: Диалог-МИФИ, 1998. – 284 с.

5. Дарнелл, Р. JavaScript: справочник / Р. Дарнелл. – СПб: Питер, 1998. – 192 с.

6. Дунаев, С. INTRANET-технологии С. Дунаев. – М.: Диалог-МИФИ, 1997. – 288 с.

7. Федоров, А. JavaScript для всех / А. Федоров – М.: Компьютер-пресс, 1998. – 384 с.

8. Айзекс, С. DynamicHTML // С. Айзекс: пер. с англ. – СПб.: BHV-Санкт-Петербург, 1998. – 496 с.

Учебное издание

Жиляк Надежда Александровна

ВВЕДЕНИЕ В JAVASCRIPT

Учебно-методическое пособие

Редактор *Ю. Д. Нежикова*
Компьютерная верстка *Е. В. Ильченко*
Корректор *Ю. Д. Нежикова*

Подписано в печать 20.10.2014. Формат 60×84 $^{1}/_{16}$.
Бумага офсетная. Гарнитура Таймс. Печать офсетная.
Усл. печ. л. 14,0. Уч-изд. л. 14,5.
Тираж 100 экз. Заказ .

Издатель и полиграфическое исполнение:
УО «Белорусский государственный технологический университет».
Свидетельство о государственной регистрации издателя,
изготовителя, распространителя печатных изданий
№ 1/227 от 20.03.2014.
ЛП № 02330/12 от 30.12.2013.
Ул. Свердлова, 13а, 220006, г. Минск.